プログラミング初心者・Webプログラマになりたい人向け

超入門テキスト

Spring Boot3で始める Webアプリケーション開発入門

菊田　英明
KIKUTA HIDEAKI

Hello，world!からマスターメンテプログラムまで
ステップ・バイ・ステップで
確実に身に付けよう

基礎編

１．はじめに

本書は**Spring Boot3**を使ってWebアプリケーション開発を始めたい方向けの入門書です。

想定している読者は

・ITシステム開発企業の新入社員

・転職してWebプログラマーになりたい人

・自分でも何かWebアプリケーションを作ってみたい人

といった方々です。

著者は長年Java言語によるシステム開発に携わってきました。近年はITシステム開発企業の新入社員向けJava導入教育の講師なども務めており、要望があればSpring Bootも教えています。ただしServlet/JSPなど従来技術をやったあと、さらにSpring Bootへチャレンジする、というスタイルです。

そのとき、ふと**「プログラミン初心者に、いきなりSpring Bootを使ったWebアプリケーション開発を教えることは可能だろうか？」**という疑問が湧きました。最近は、導入教育もそこそこに、開発現場に投入される人の話も聞きます。それは極端な例としても、「最初の仕事がSpring BootでWebアプリケーション開発」というパターンはありえるでしょう。本書はそういった方を念頭において執筆しました。

Spring Boot 3

Spring Boot 3(https://spring.io/projects/spring-boot)は**Spring Framework**というフレームワーク群(https://spring.io/projects)をシンプルに活用しやすくしたものです。Spring Frameworkを直接使うより少ない手間でWebアプリケーションを構築できます。

このSpring Boot 3はSpring Bootの3番目のメジャーバージョンであり、2022年11月にリリースされました。それ以前のメジャーバージョンであるSpring Boot 2から一部のパッケージ名が変更されるなど、ソースコード的に非互換な部分があります。

本書はこの最新のSpring Boot 3を使ってWebアプリケーションを作っていきます。

なお本書ではSpring Boot3をSpring Bootと表記します。

本書の目標

この本を読み終わったら**「簡単なマスタメンテプログラムを作れる」**ようになるのが目標です。

「マスタ」は、そのシステムが管理する顧客や商品など基本的なデータを集めたものです。このマスタに対して検索、追加、更新、削除といった操作を行えるようにするのが「マスタメンテプログラム」であり、実際のシステム開発でも頻繁に登場します。これをSpring Bootを使ったWebアプリケーションとして作成するにはどういった知識が必要か？そこから逆算して作成したのが本書です。

本書の内容は、上記目標を実現するのに必要最低限な事項を中心としています。その上で、重要ポイントは冗長にならない程度に繰り返し説明しています。

一方「Spring Framework」「DI(依存性注入)」「MVC(モデル-ビュー-コントローラ)パターン」などについては、言及していません。プログラミング初期段階で、こういった概念的な話題を説明しても「有用性がイメージしにくく、腹落ちするのが難しい」という導入教育での経験によるものです。本書を読み終わってから学んだ方が理解しやすいでしょう。

前提知識

本書を読み進めるには「Java言語」「HTML」「リレーショナルデータベース/SQL」の知識が必要です。おおよそ以下のようなレベルを前提としています。

・Java言語

クラス、インタフェース、制御文に加え、Listなどのコレクションやジェネリックスがある程度わかればベストです。

・HTML

ブラウザに「Hello, world!」といった文字列を表示させた経験があればOKです。<table><th><tr><td>タグやリンク(<a href='xxx'>)を知っていれば最高です。それ以外に必要なタグは本書で説明します。

・リレーショナルデータベース/SQL

「リレーショナルデータベースのテーブルは、表のようなもの」といった基礎知識を持っているのが望ましいです。また簡単なSQL文(SELECT,INSERT,UPDATE, DELETE)がわかれば申し分ないです。

本書の構成

本書の構成は下表のようになっています。

章	タイトル	備考
1章	はじめに	本章
2章	環境設定	
3章	Spring Bootで Hello, world	Webアプリケーションの基礎知識
4章	Thymeleafでフォーム操作	〃
5章	セッション操作	〃
6章	テーブルのデータを一覧表示する	ToDo管理アプリケーションの開発
7章	テーブルにレコードを追加する	〃
8章	テーブルのレコードを更新・削除する	〃
9章	入力された条件で検索する	〃
10章	動的なクエリによる検索	〃
11章	ページネーション(ページング)	〃
ー	演習課題	
ー	参考情報	

2章で開発環境を準備し、3～5章でWebアプリケーションの基礎を説明します。また5章では、初心者にとって「継承」以上に鬼門と思える「参照」について、パラパラ漫画風の図を多用して徹底解説します。

6～10章では「ToDo管理アプリケーション」をステップ・バイ・ステップで開発していきます。このToDoを一種のマスタと読み替えれば、マスタメンテプログラムを作成できます。さらに11章では(類書では見かけないが)実務では必ずと言っていいほど必要になる「動的クエリのページネーション(ページング)」についても解説します。

以上の内容を理解できれば、現場で周りの人との会話にも、ある程度ついていけるようになるでしょう。また類書やインターネット上の解説記事を読みこなすこともできるようになるでしょう。

本書の進め方

本書では各章の冒頭で何をやるか、何を作るかを説明します。あわせて画面のスクリーンショットやUMLのシーケンス図を提示します。そのあとプログラムをできるだけ細かく解説します。

もし時間に余裕があれば、掲載しているプログラムを手入力しながら読み進めることをお勧めします。理由は2つあります。

1つは、開発ツールに慣れるためです。

本書ではSTSという、Spring Bootに欠かせない開発ツールを使います。操作方法は本書でも説明しますが、これを利用し3章と4章では(非常に単純ですが)7個のWebアプリケーションを作ります。短い

サイクルで繰り返し手を動かせば、それだけ早くSTSを使いこなせるようになります。実務では、こういった開発ツールに対する習熟度が生産性の差として現れる場合があります。その最初のチャレンジだと思ってSTSに取り組んでください。

2つめは、プログラムを手入力していくと、より深く学習できるからです。

書いてあるとおり入力したつもりでも、文法エラーや実行時エラーが発生するでしょう。それを解決すればするほど経験値がアップします。それ以外にもたくさんの気づきがあるでしょう。「なぜこの値(名前)を使うのだろう？」「あそことここは、何が違っているのだろう？」といった疑問はその典型です。それを自分で考える、周りの人に聞く、あるいはググるなどして、クリアしていくことが、本当の学習であると著者は考えています。

人は理論ではなく、例題から学びます。だから「論よりラン(run;実行)」です。手を動かしながら読み進めてください。

サポートサイト

本書掲載のプログラムは以下のURLから入手できます。追加情報があれば、あわせて掲載します。

https://kktworks.github.io/

kktworks@gmail.com(お問い合わせ)

@kktworks1(Twitter)

2．環境設定

本書で使用している環境を下表に示します。いずれも2022年11月時点の安定版です。実際にインストールするものとは多少バージョンが異なると思いますが、適宜読み替えてください。

【表2-1】本書の前提環境

No.	名称	Ver	機能	備考
1	Windows10(64bit)	-	-	
2	Chrome	107.0	Webブラウザ	
3	Eclipse Temurin JDK	17.0.5+8	Javaプログラム開発キット (Java Development Kit)	No.4 の前提ソフト
4	Spring Tool Suite 4	4.16.1	Spring Bootアプリ開発環境	Spring Boot 3.0.0
5	Eclipse Web開発ツール	3.27	HTML/CSS編集用プラグイン	
6	Lombok	1.18.24	Java定型コード生成ツール	
7	PostgreSQL	15.1	リレーショナルデータベース 管理システム	

以下No.3～7のインストール方法を説明します。

2.1 Eclipse Temurin JDK

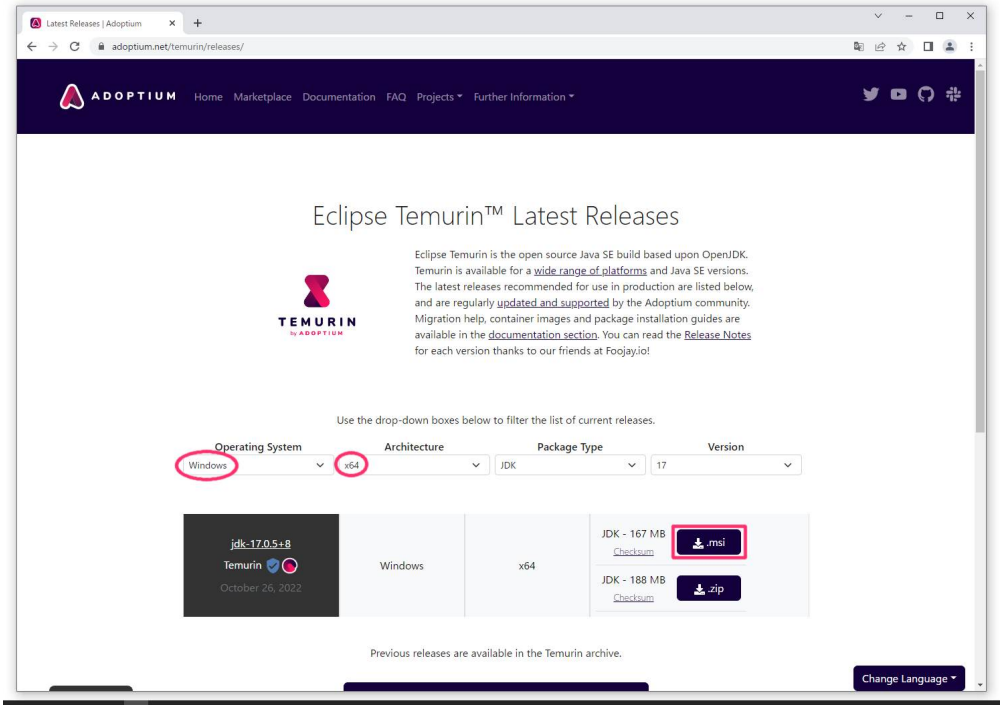
JDKには開発元の違いなどにより、複数の種類(ディストリビューション)が存在します。たとえばSpring Bootがベースにしているフレームワークは「BellSoft Liberica JDK」というものを推奨しています(https://spring.io/quickstart)。しかし本書ではよりメジャーなJDKであったAdoptOpenJDKの流れを組む「Eclipse Temurin JDK」を使用します。

インストール手順

1)ダウンロード

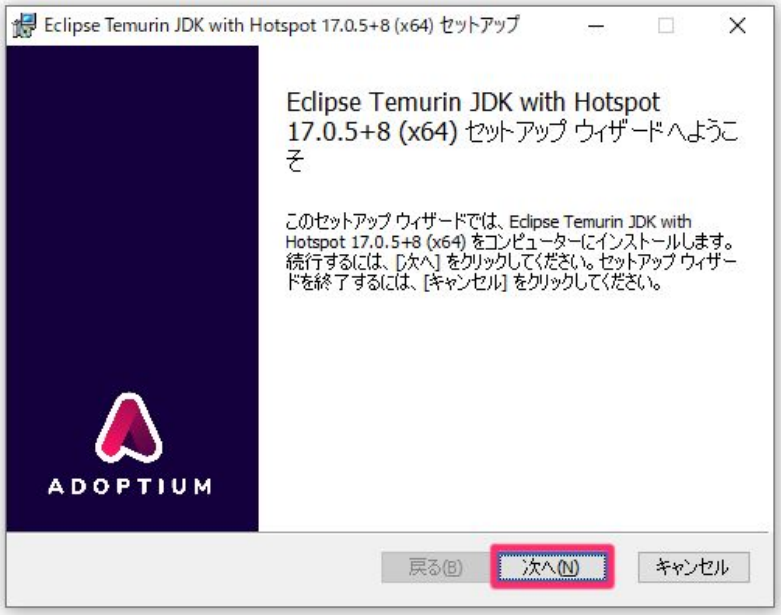
ブラウザでhttps://adoptium.net/temurin/releases/を開く。

Operating System[Windows], Architecuture[x64]を選択し、[.msi]をクリックする ⇒ ダウンロードが始まる



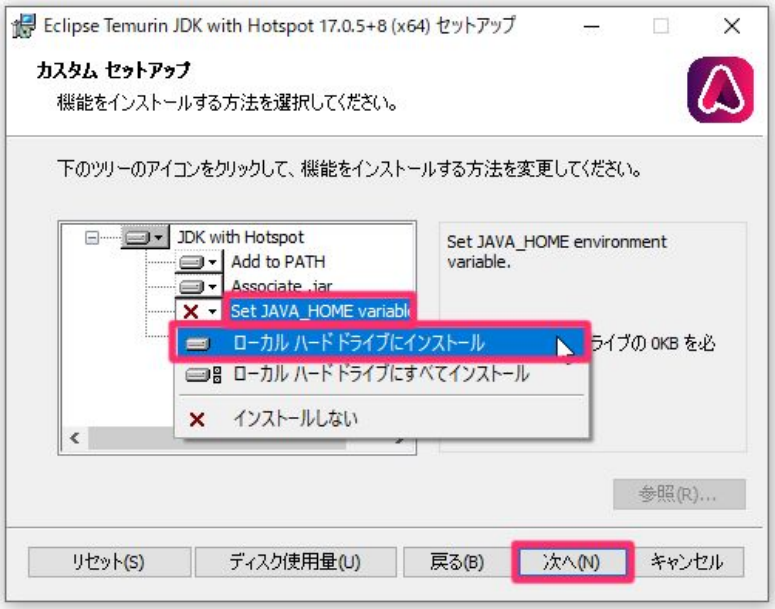
2)ダウンロードしたファイル(msiファイル)を、エクスプローラーでダブルクリックして実行する。

3)セットアップウィザードが起動する ⇒ [次へ(N)]ボタンをクリックする。

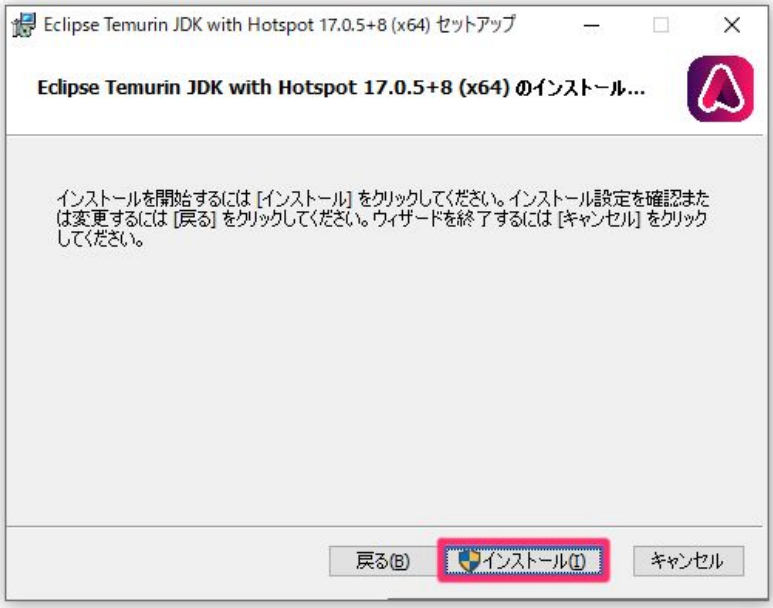


4)使用許諾契約書が表示された場合は⇒ 内容確認 > [使用許諾契約書に同意します]をチェック > [次へ(N)]ボタンをクリックする。

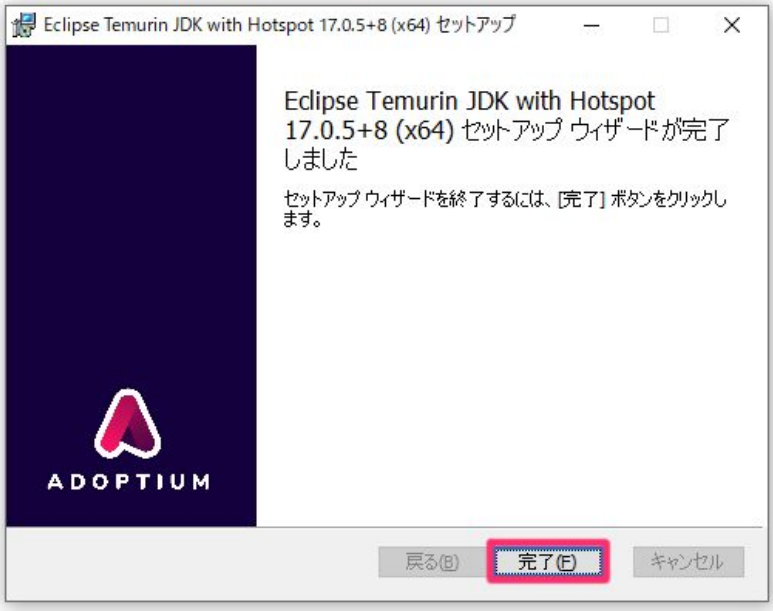
5)インストール機能選択画面が表示される。
[Set JAVA_HOME variable]をクリック > [ローカル ハード ドライブにインストール]を選択 > [次へ(N)]ボタンをクリックする。



6)[インストール(I)]ボタンをクリックする。



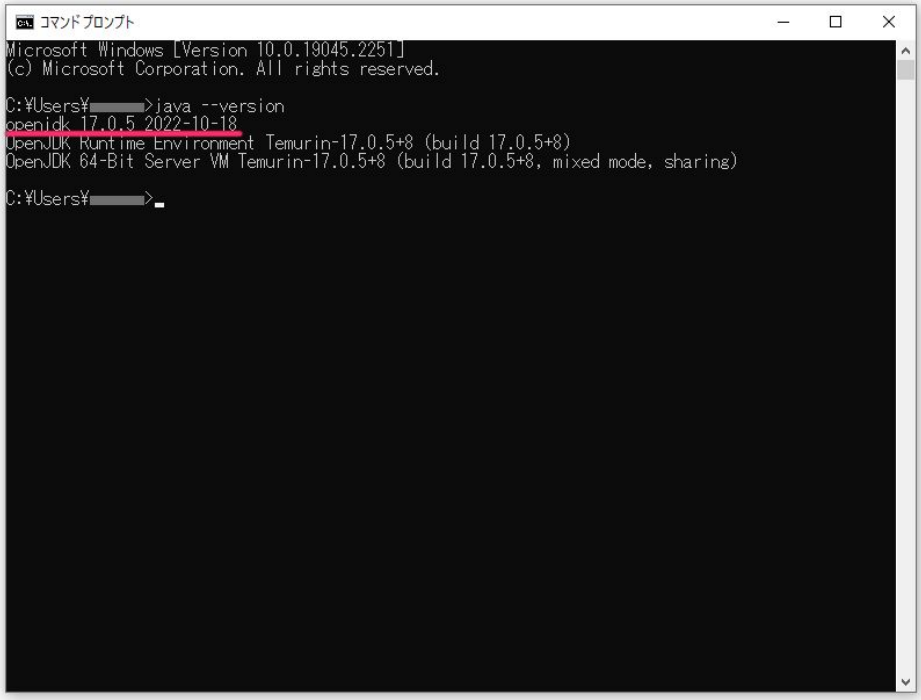
7)完了したら[完了(F)]ボタンをクリックする。



8)正常にインストールされたことを確認する。
　コマンドプロンプトを開き、以下のコマンドを実行する。

```
java  --version
```

⇒ Versionが表示されることを確認する。



⇒ exitと入力するか、右上の[X]をクリックしてコマンドプロンプトを閉じる。

2.2 Spring Tool Suite 4

Spring Bootを使ったWebアプリケーション開発には、Spring Frameworkの開発元であるSpringSourceが提供するツールを利用するのが一般的です。その方法には、大きく分けて次の２つがあります。

1)Eclipse, Visual Studio Codeなどの統合開発環境(IDE)に「Spring Tools 4」を追加インストールする。
2)「Spring Tool Suite 4(以下STS)」を使用する。

本書では2)のSTSを使います。STSは「Spring Tools 4が組み込まれたEclipse」であり、セットアップが簡単です。使用するPCにEclipseがインストールされていても、STSを別フォルダにすれば共存できます。

インストールしたSTSはPleiadesプラグインにより日本語化します。

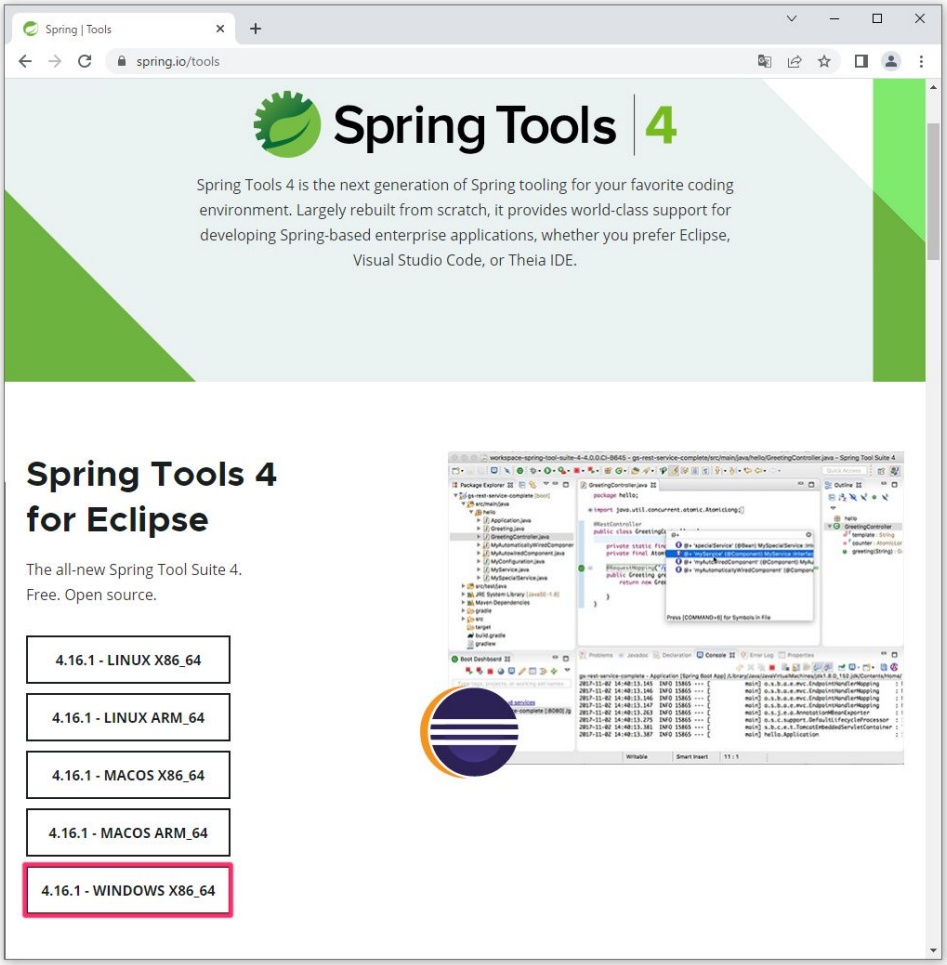
2.2.1 STSのインストール

インストール手順

1)ダウンロード

ブラウザでhttps://spring.io/tools/を開く。

Spring Tools 4 for Eclipseの[WINDOWS X86_64]をクリックする ⇒ ダウンロードが始まる

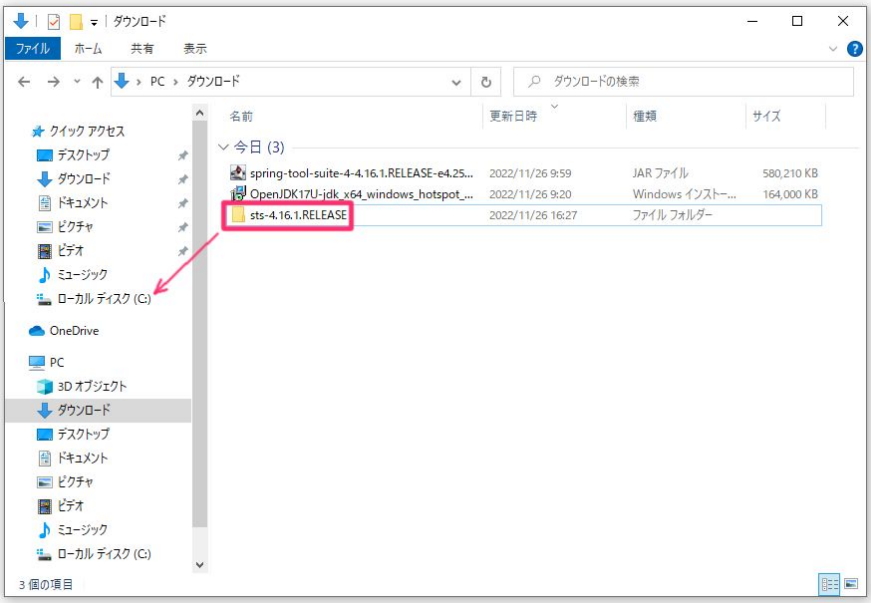


2)ダウンロードしたファイル(jarファイル)をエクスプローラーでダブルクリックする ⇒ 自動的に解凍される

3)解凍されたフォルダ(下図ではsts-4.16.1.RELEASE)を、任意の場所に移動させる。

⇒ 本書ではC:\下へ移動させたものとします。

⇒ 以降このC:\sts-4.16.1.RELEASEを**<STS_DIR>**と表記します。



2.2.2 STSの日本語化

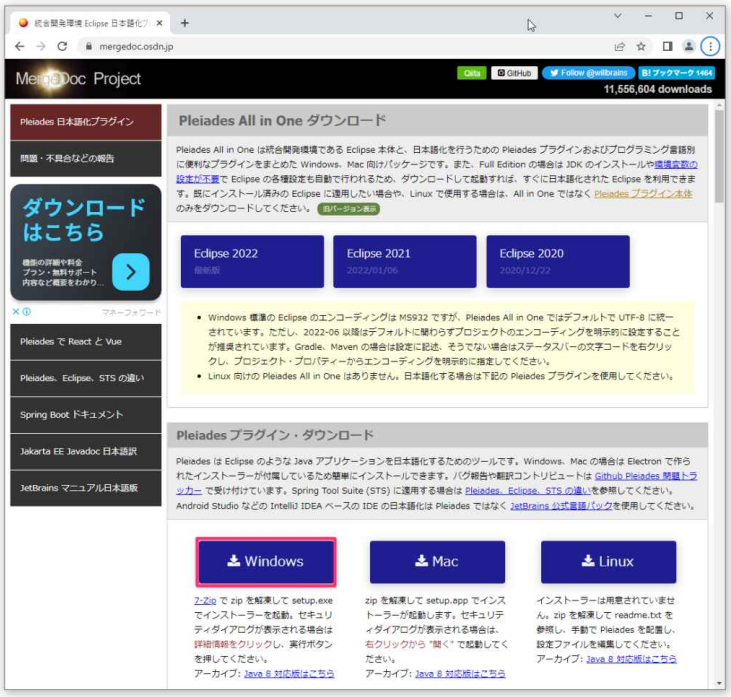
インストールしたSTSをPleiadesプラグインにより日本語化します。

日本語化手順

1)ダウンロード

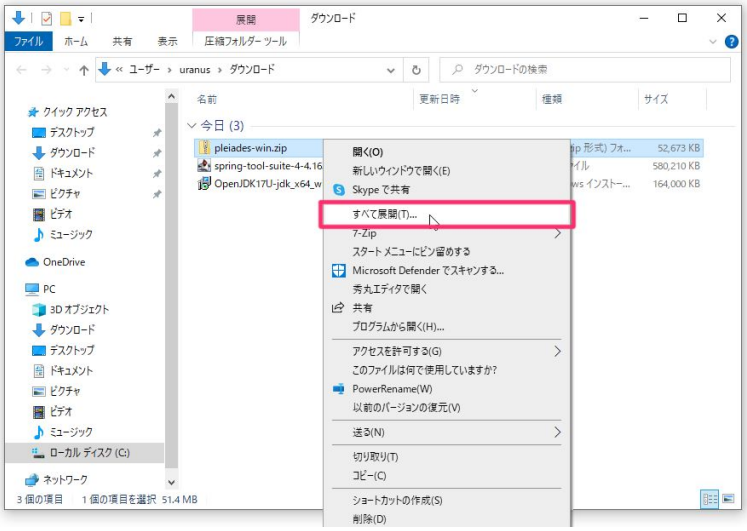
ブラウザでhttps://mergedoc.osdn.jp/を開く。

画面下部「Pleiadesプラグイン・ダウンロード」にある[Windows]をクリックする。

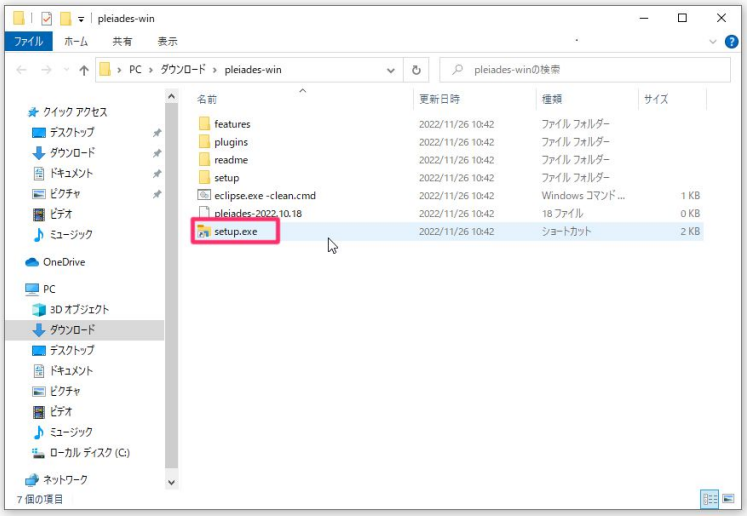


2)ダウンロードリンクをクリックする。

3)ダウンロードしたファイル(zipファイル)をエクスプローラーで右クリック ›
[すべて展開(T)...]を選択し、解凍する。

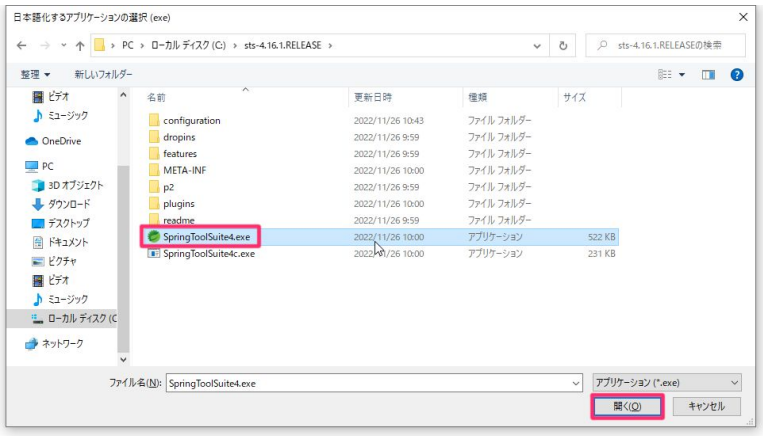


4)解凍したフォルダの中にあるsetup.exeをダブルクリックして起動する。

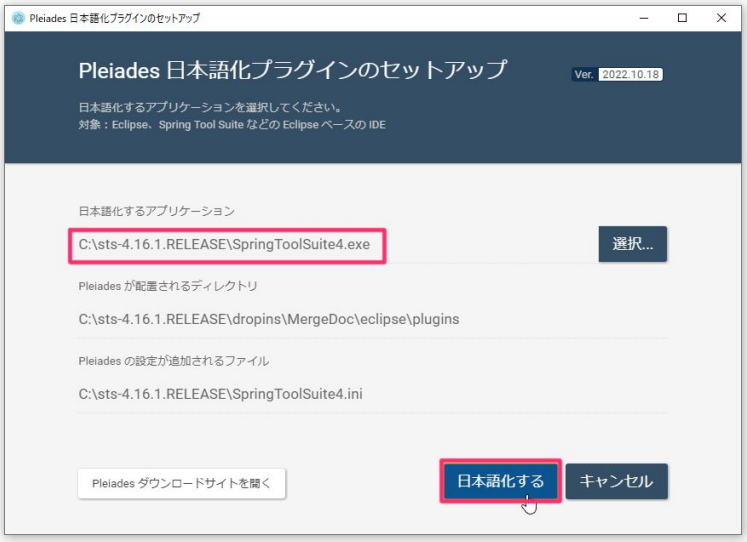


5)セットアップ画面が表示される ⇒ [選択...]ボタンをクリックする。

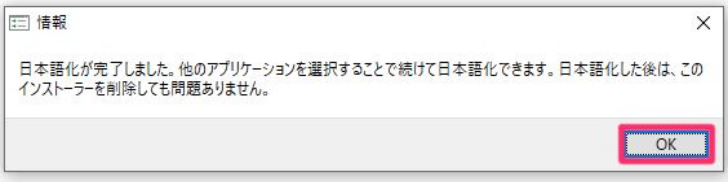
6)<STS_DIR>下のSpringToolSuite4.exeを選択 > [開く(O)]ボタンをクリックする。



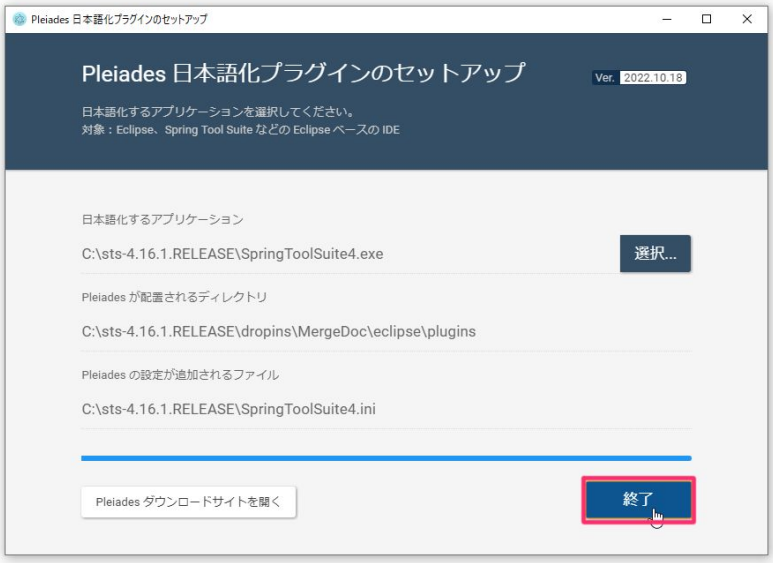
7)「日本語化するアプリケーション」が6)で選択した内容であることを確認 > [日本語化する]ボタンをクリックする。



8)[OK]ボタンをクリックする。

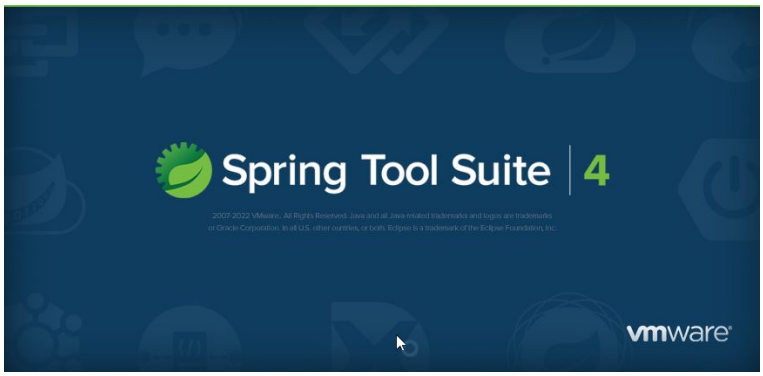


9)[終了]ボタンが表示されたらクリックする。

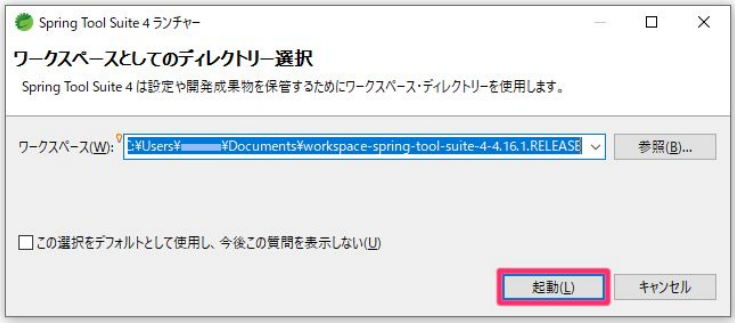


10)STSが日本語化されたことを確認する。

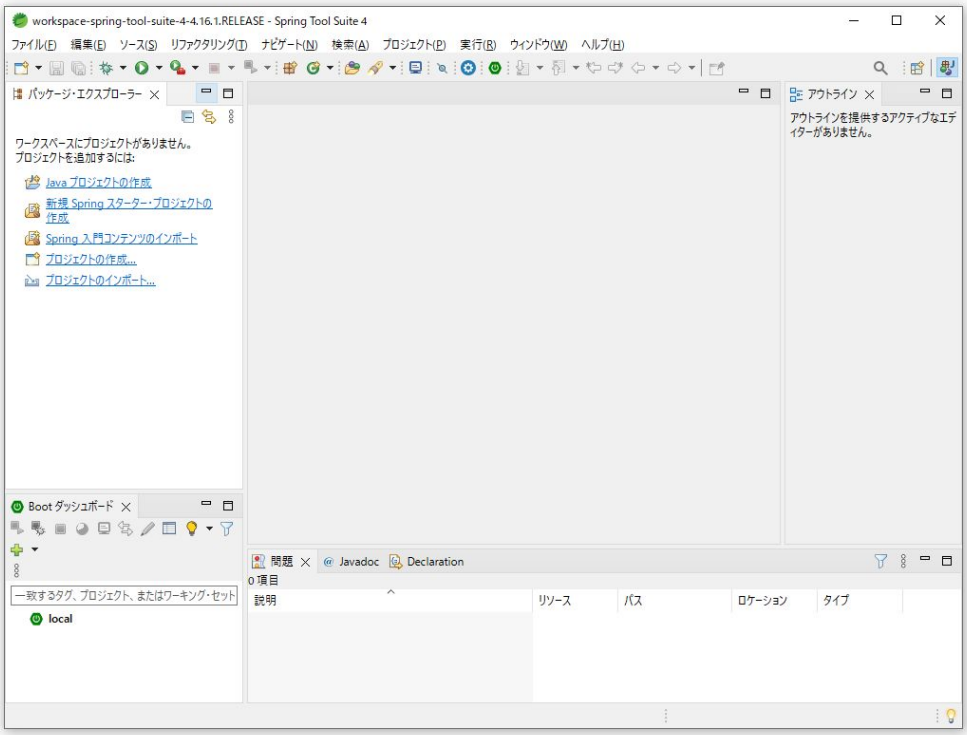
エクスプローラーで<STS_DIR>\SpringToolSuite4.exeをダブルクリックして起動する。
スプラッシュウィンドウが表示された後...



11)「ディレクトリー選択」ダイアログが表示される ⇒ [起動(L)]ボタンをクリックする。



12)STSが起動する。
メニューなどが日本語で表示されていることを確認する。

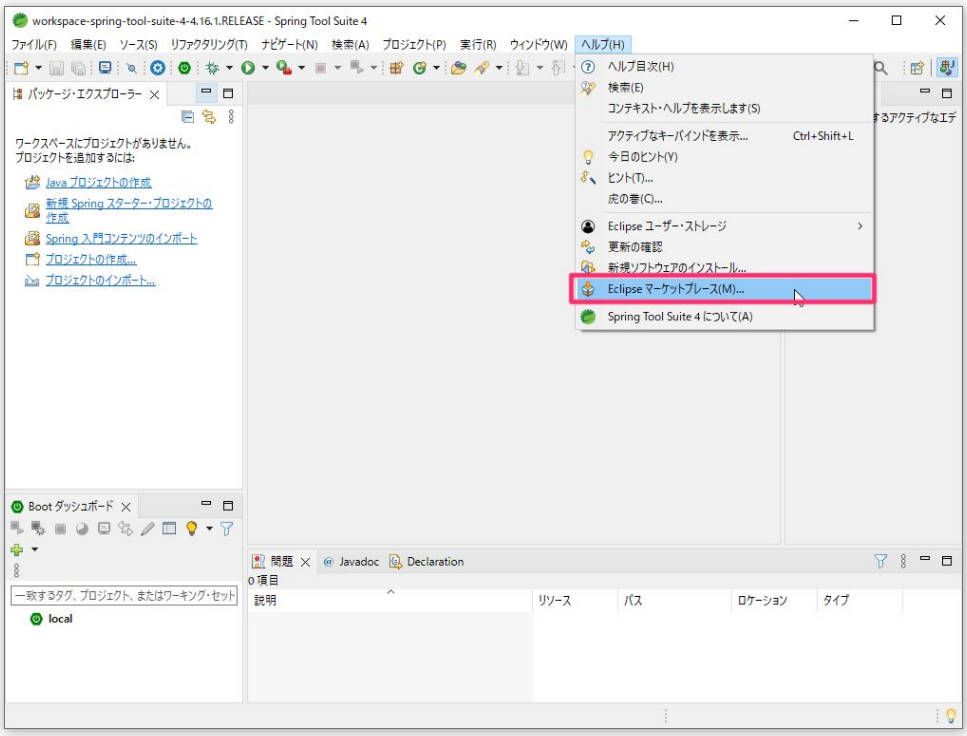


2.2.3 Web開発用プラグインのインストール

続けてSTSへHTML/CSSを編集するためのプラグインを追加します。

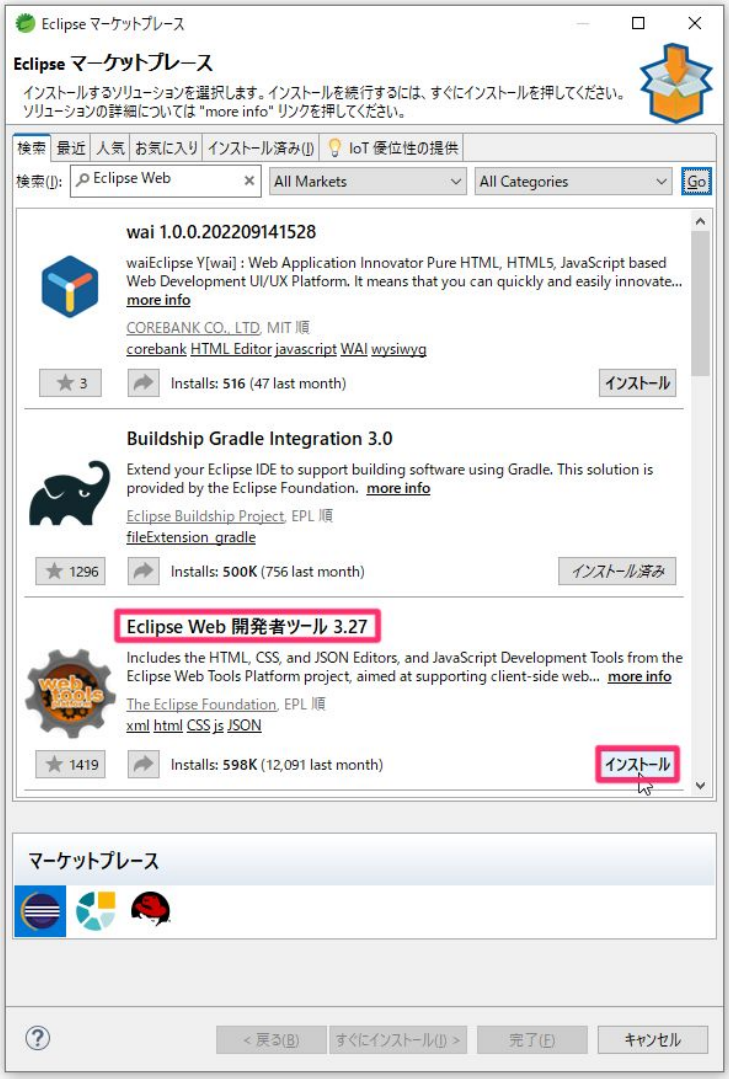
追加手順

1)STSのメニュー[ヘルプ(H)] > [Eclipse マーケットプレース(M)...]をクリックする。

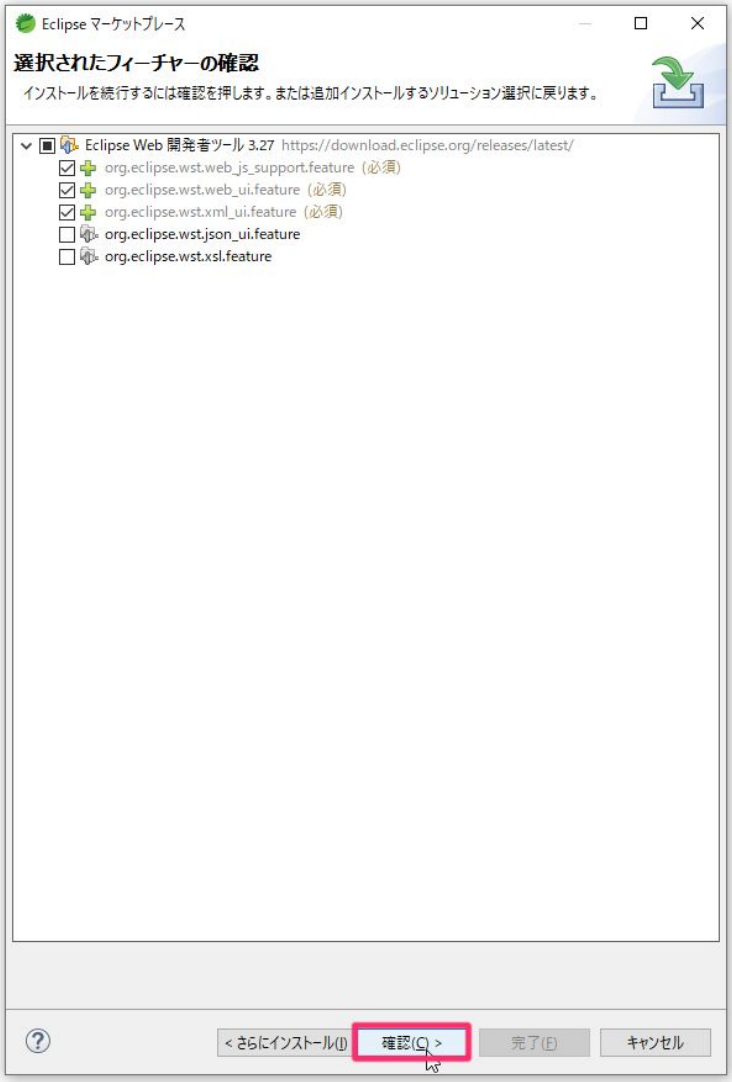


2)検索(I)欄に"**Eclipse Web**"と入力 > [Go]ボタンをクリックする。

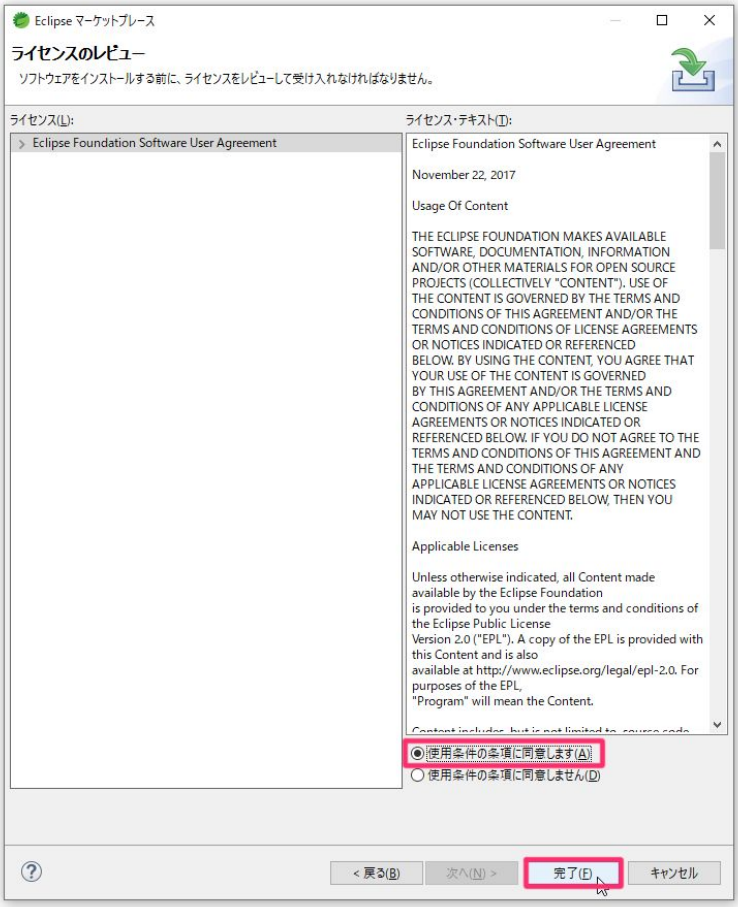
3)表示された“Eclipse Web 開発者ツール”の[インストール]ボタンをクリックする。



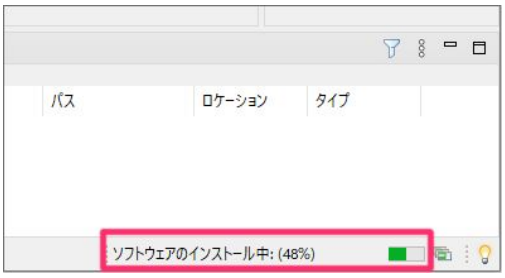
4)必須機能がチェックされていることを確認 > [確認(C)]ボタンをクリックする。



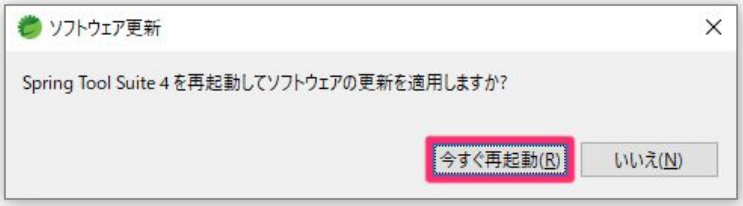
5)ライセンスを確認 > [使用条件の条項に同意します(A)]を選択 > [完了(F)]ボタンをクリックする。



6)STSのウィンドウ右下にインストール状況が表示される。



7)完了したら[今すぐ再起動(R)]ボタンをクリックする。



8)「終了の確認」ダイアログが表示された場合は、[終了(E)]ボタンをクリックする。

9)ショートカットの作成(オプション)

STSの設定は以上です。

<STS_DIR>\SpringToolSuite4.exeのショートカットをデスクトップに作成しておきましょう。

名称は適宜変更してください。

「スタートメニューにピン留めをする」「タスクバーにピン留め」するでも構いません。STSを起動しやすくしておきましょう。

2.3 Lombok

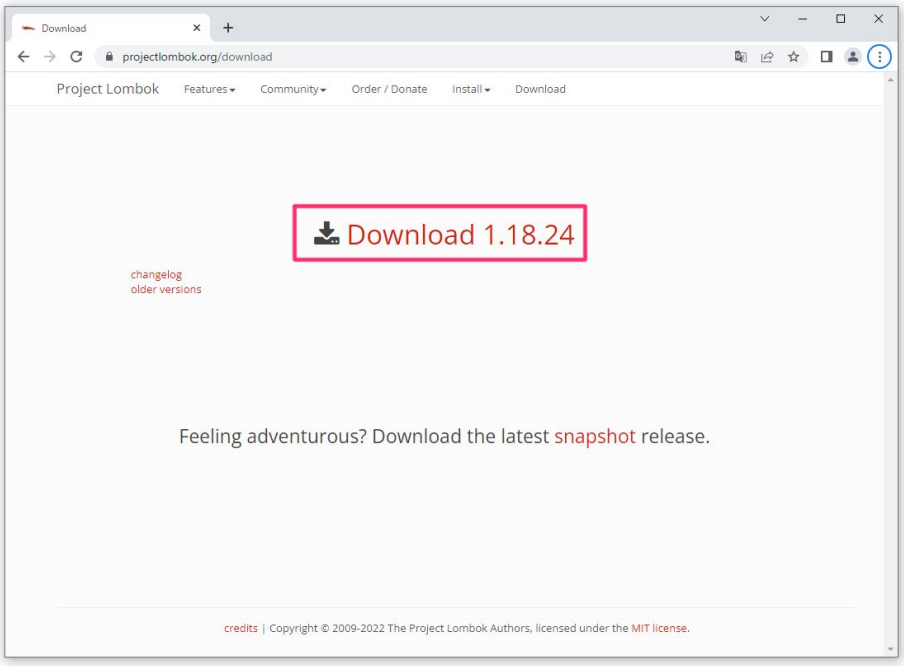
Javaでプログラムを作成しているとsetXXX()やgetXXX()など、同じようなコードが繰り返し現れます。こういった定型的なものはSTSで自動生成できますが、Lombok(ロンボック/ロンボク)を使えば、より効率的に対処できます。

インストール手順

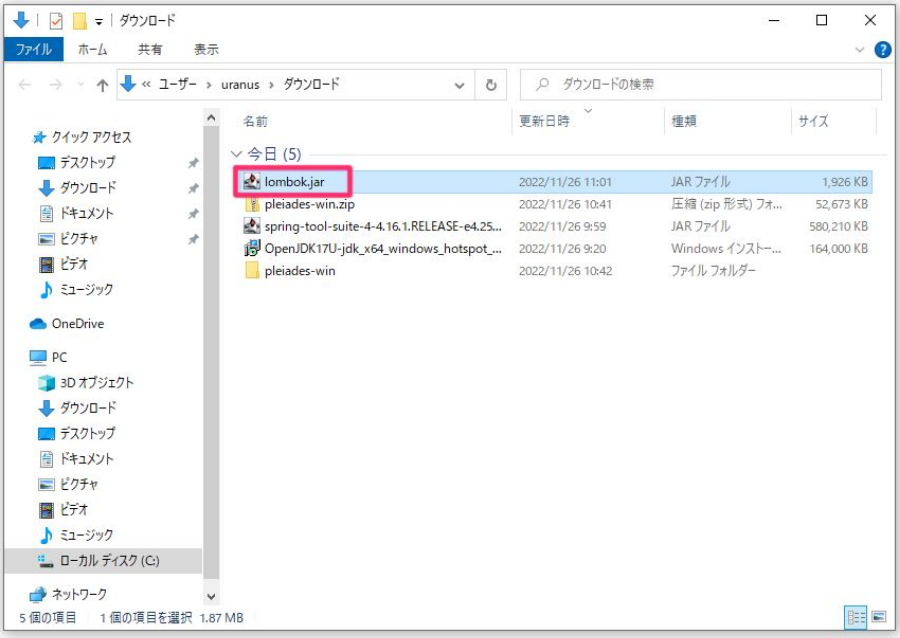
1)ダウンロード

ブラウザでhttps://projectlombok.org/downloadを開く。

[Download x.xx.xx]の部分をクリックする ⇒ ダウンロードが始まる



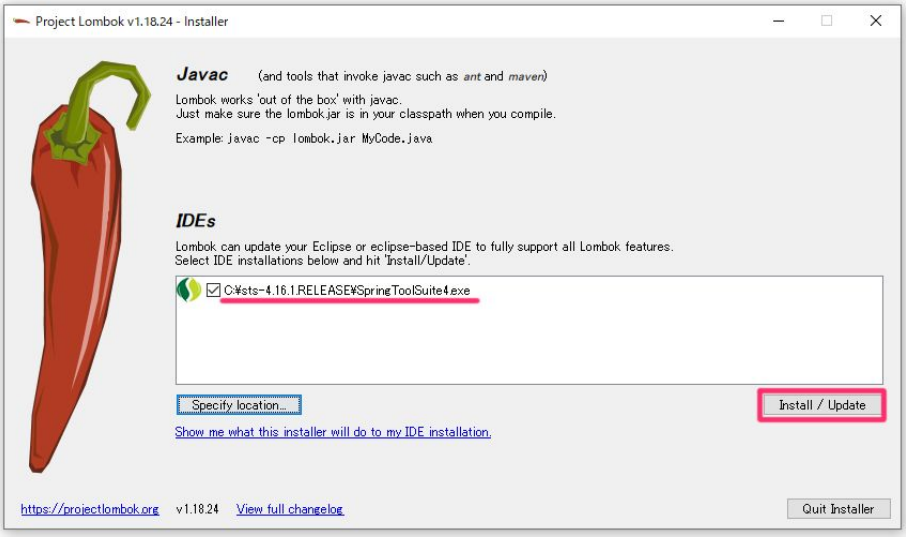
2)ダウンロードしたファイル(jarファイル)をエクスプローラーでダブルクリックして実行する。



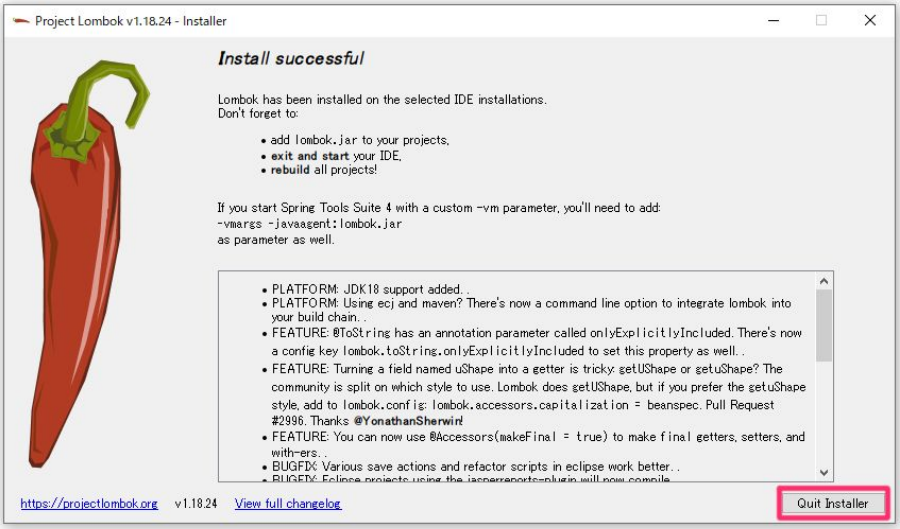
3)インストーラ–が起動し、STSがインストールされている場所を探す。

⇒ [IDEs]に<STS_DIR>\SpringToolSuite4.exeが表示されるのを待つ。

⇒ チェックされていることを確認し[Install/Update]ボタンをクリックする。

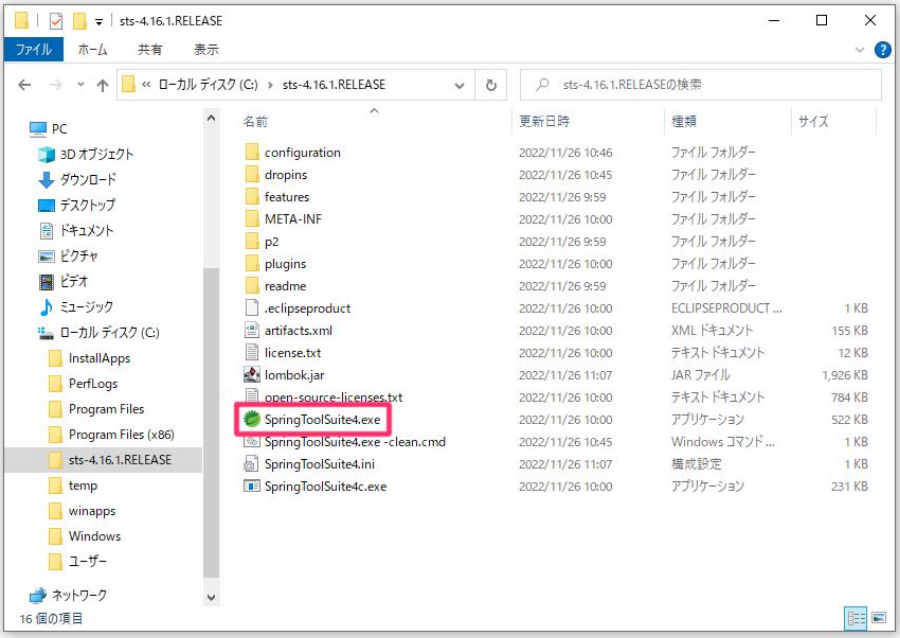


4)インストールが完了したら[Quit Installer]ボタンをクリックする。



5)Lombokが正常にインストールされたことを確認する。

⇒ <STS_DIR>にlombok.jarがあることを確認する。



2.4 PostgreSQL

本書ではリレーショナルデータベース管理システム(RDBMS)として、PostgreSQL(ポストグレスキューエル)を使います。

Spring Bootにも「H2」というデータベースが組み込まれており、設定ファイルを用意するだけでプログラムからアクセスできます。しかし純粋にデータベースやSQLを学びたいと思ったときには、かなりの不便を強いられます。

データベースシステムはSpring Bootから独立したものを準備した方が良いでしょう。その方が後々学習の幅を広げやすいでしょう。

そこで本書では、商用製品に匹敵する機能と性能を持ち、業務システムでも多くの使用実績があるオープンソースのPostgreSQLを使います。

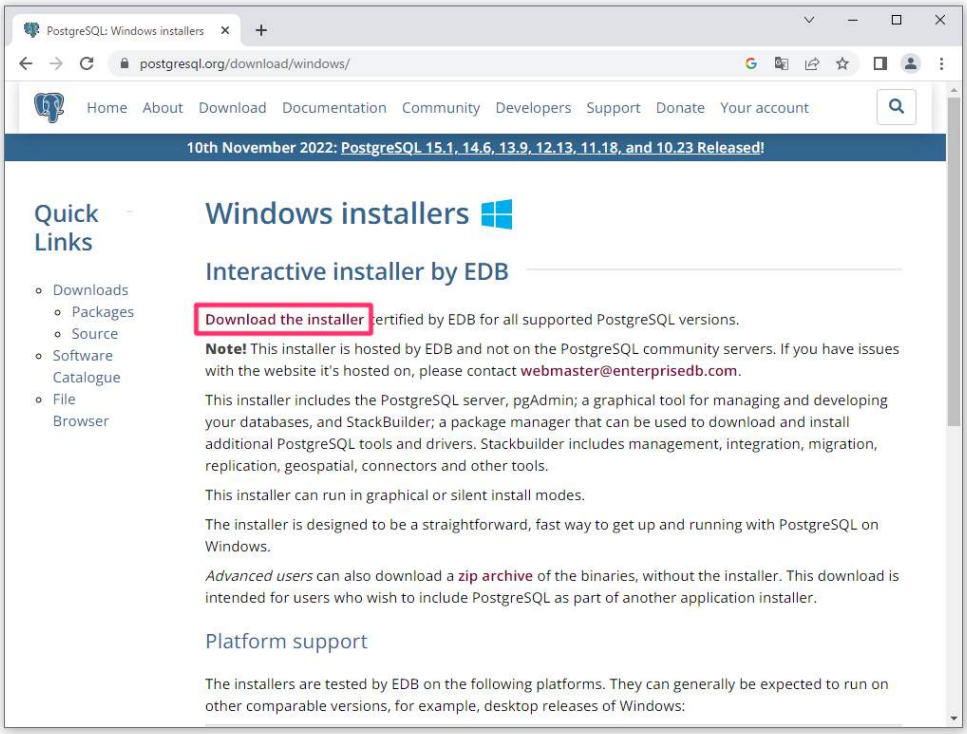
PostgreSQLはインストールしたあと、環境変数を設定します。

2.4.1 PostgreSQLのインストール

インストール手順

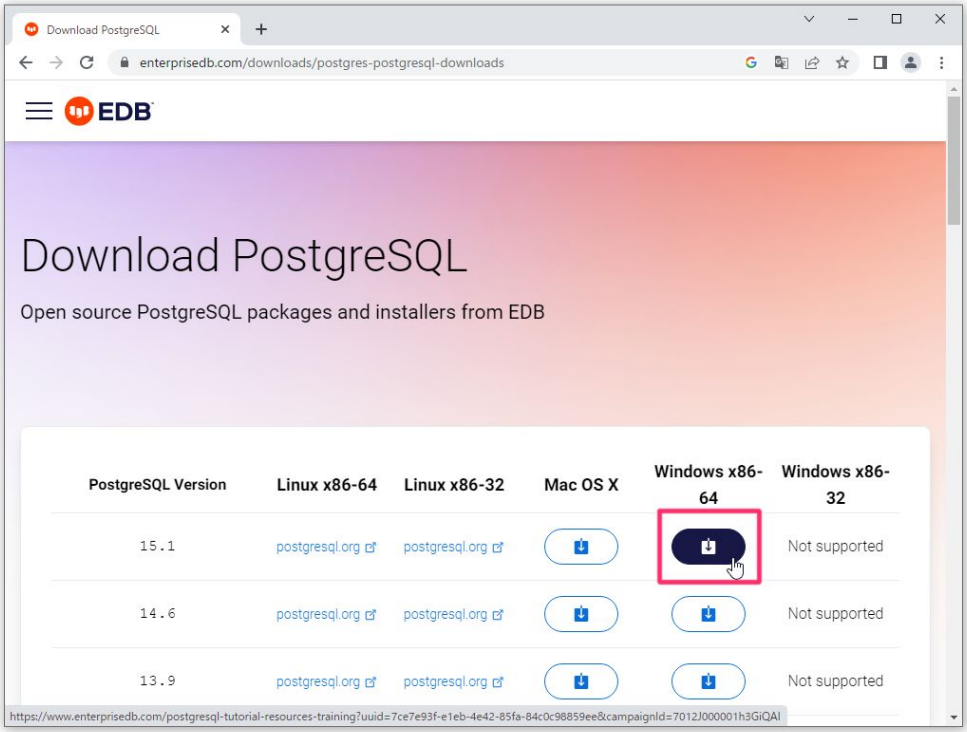
1)ブラウザでhttps://www.postgresql.org/download/windows/を開く。

⇒ "Download the installer"をクリックする。



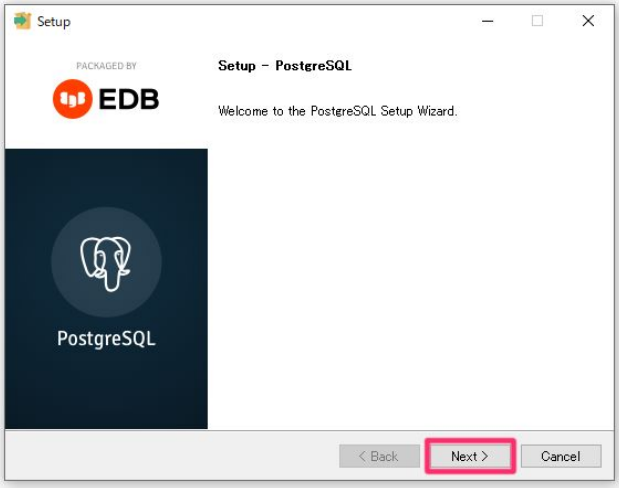
2)[↓]をクリックする ⇒ ダウンロードが始まる

本書では64ビットWindows用Version 15.1を使用します。

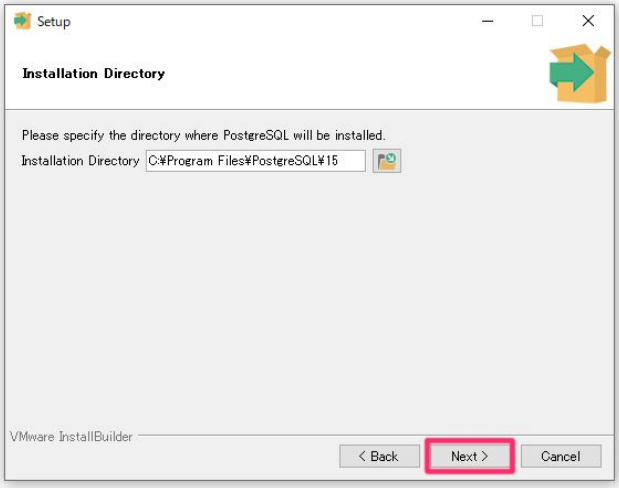


3)ダウンロードしたファイル(exeファイル)をエクスプローラーでダブルクリックして起動する。

4)セットアップウィザードが表示される ⇒ [Next>]ボタンをクリックする。

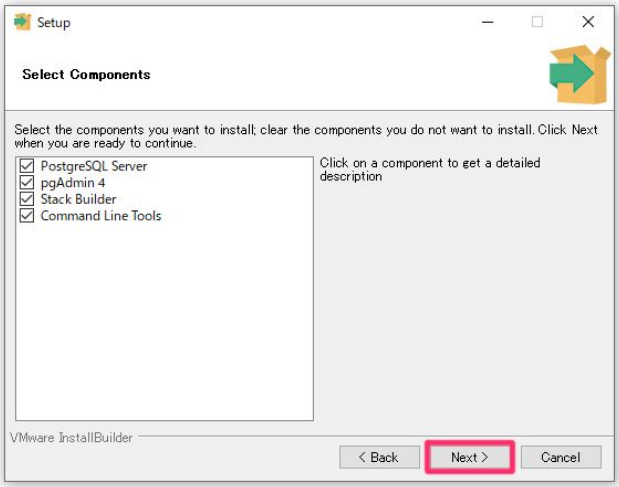


5)インストール先選択画面が表示される。
⇒ [Next]ボタンをクリックする(本書ではデフォルトのC:\Program Files\PostgreSQL\15のままとします)。
⇒ 以下、このインストール先を**<POSTGRES_DIR>**と表記します。



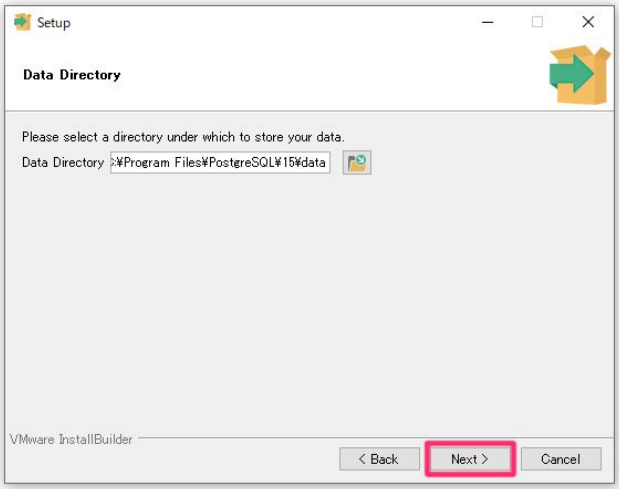
6)インストール機能選択画面が表示される。

⇒ [Next>]ボタンをクリックする(本書ではデフォルトのままとします)。



7)データベースファイル格納位置選択画面が表示される。

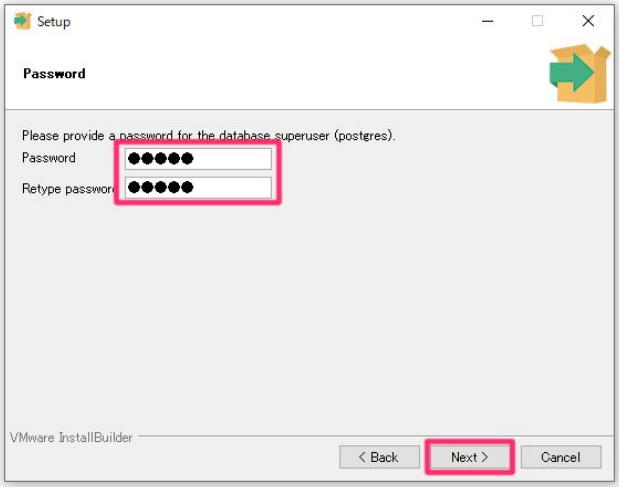
⇒ [Next>]ボタンをクリックする(本書ではデフォルトのままとします)。



8)PostgreSQL管理者のパスワード入力画面が表示される。

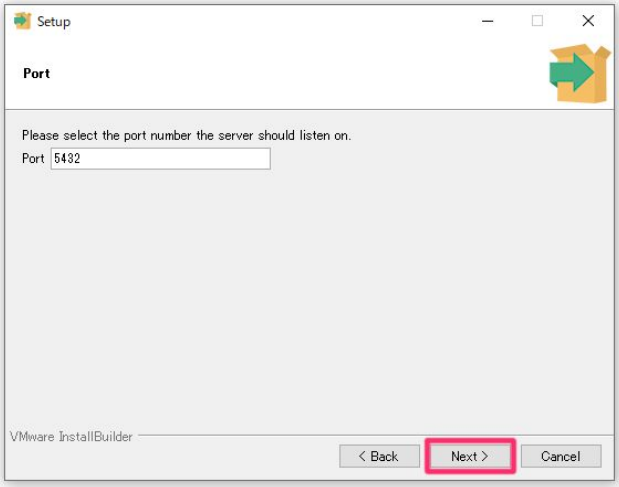
⇒ 適宜パスワードを入力し、[Next>]ボタンをクリックする。

※ここで入力したパスワードは6章で使います。忘れないこと！



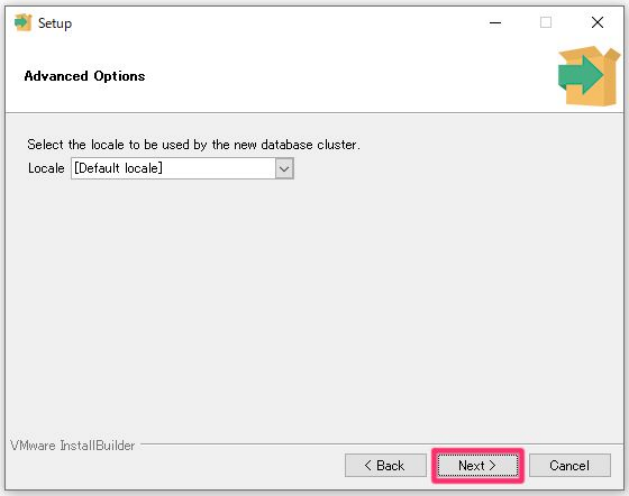
9)Port番号入力画面が表示される。

⇒ デフォルトの"5432"のままとする。[Next>]ボタンをクリックする。

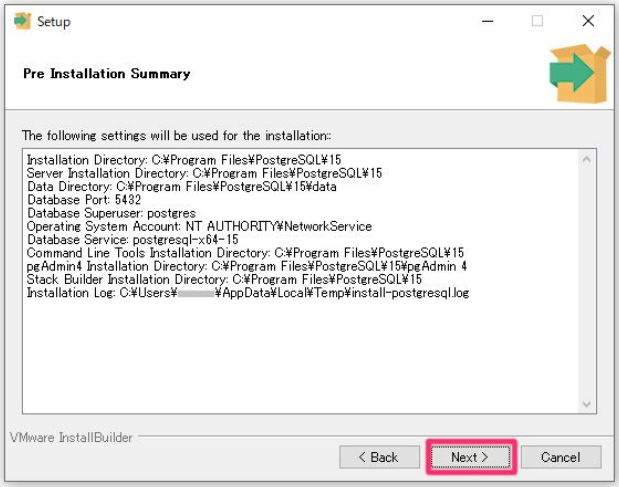


10)Locale入力画面が表示される。

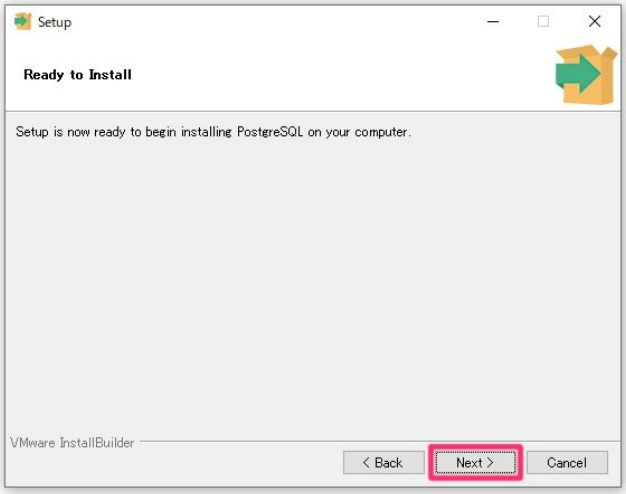
⇒ デフォルトの"Default Locale"のままとする。[Next>]ボタンをクリックする。



11)インストール条件が表示される ⇒ [Next>]ボタンをクリックする。

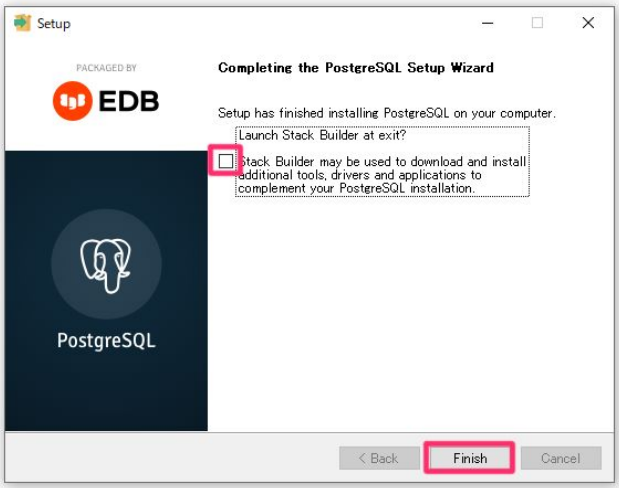


12)[Next>]ボタンをクリックする ⇒ インストール開始



13)インストール完了

⇒ 「Lunch Stack Builder are exit?」を**クリア** > [Finish]ボタンをクリックする。



2.4.2 環境変数PATHの設定

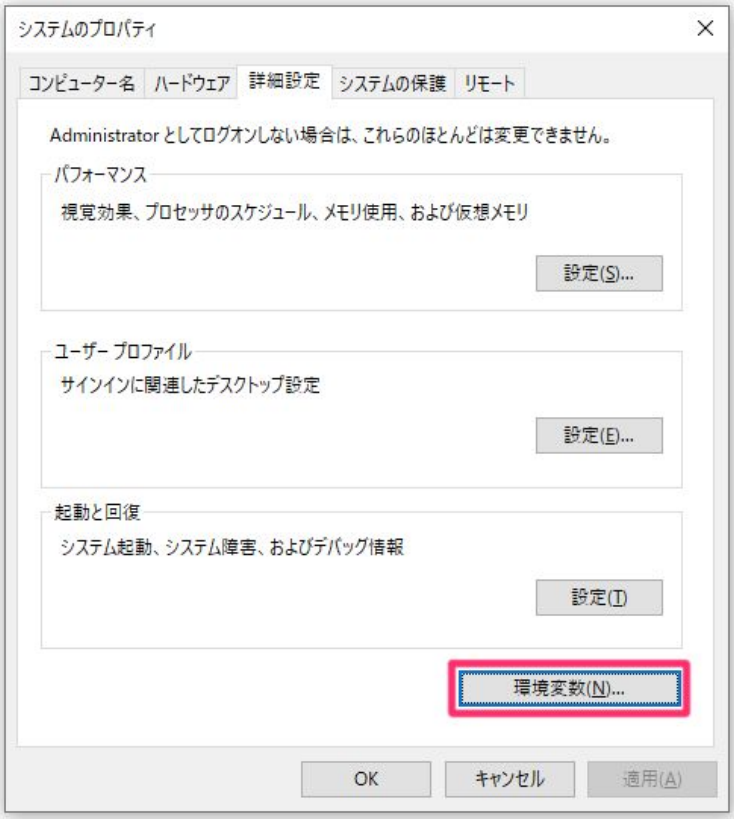
1)「システムの詳細設定」を表示する。

■表示方法(複数あります)

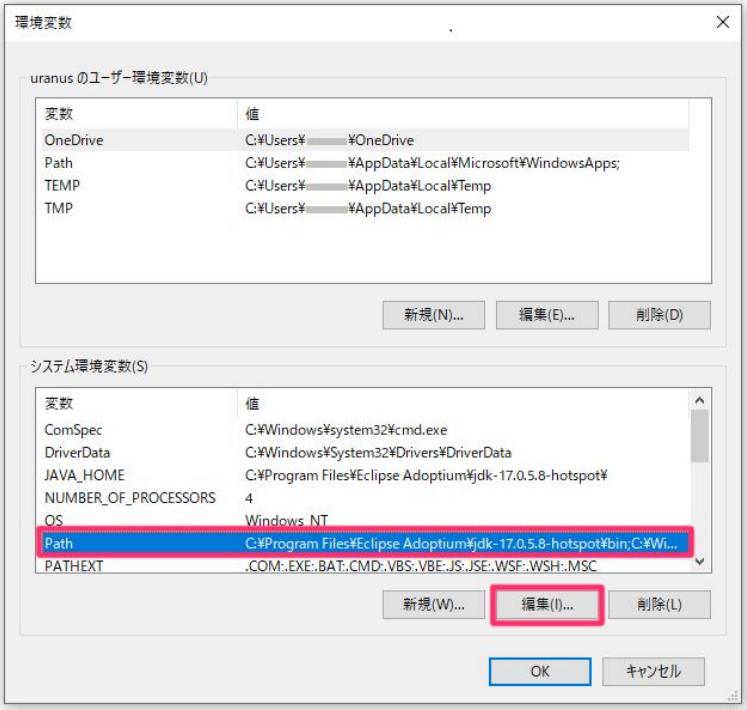
・スタートメニューを右クリック > システム > 設定ウィンドウの[設定の検索]に"詳細設定"と入力 >

候補に「システムの詳細設定の表示」が表示されるのでこれをクリック
・コントロールパネル > システムとセキュリティ > システム > システムの詳細設定(表示方法が「カテゴリ」)
・コントロールパネル > システム > システムの詳細設定(表示方法が「大きいアイコン」「小さいアイコン」)

2)[環境変数(N)...]クリックする。

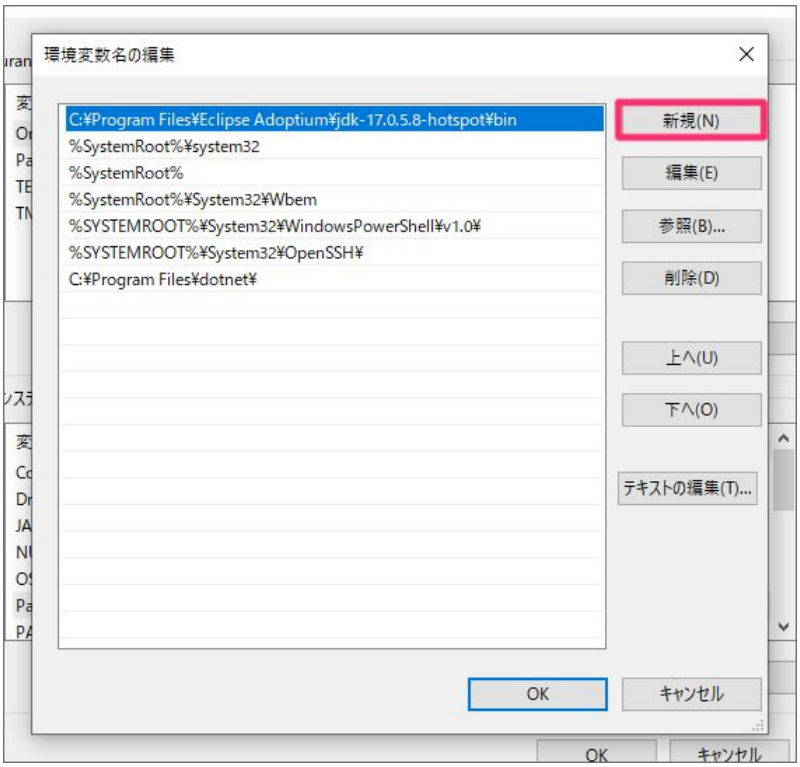


3)「環境変数」ダイアログが表示される。
⇒ システム環境変数の"Path"をクリック > [編集(I)...]ボタンをクリックする。

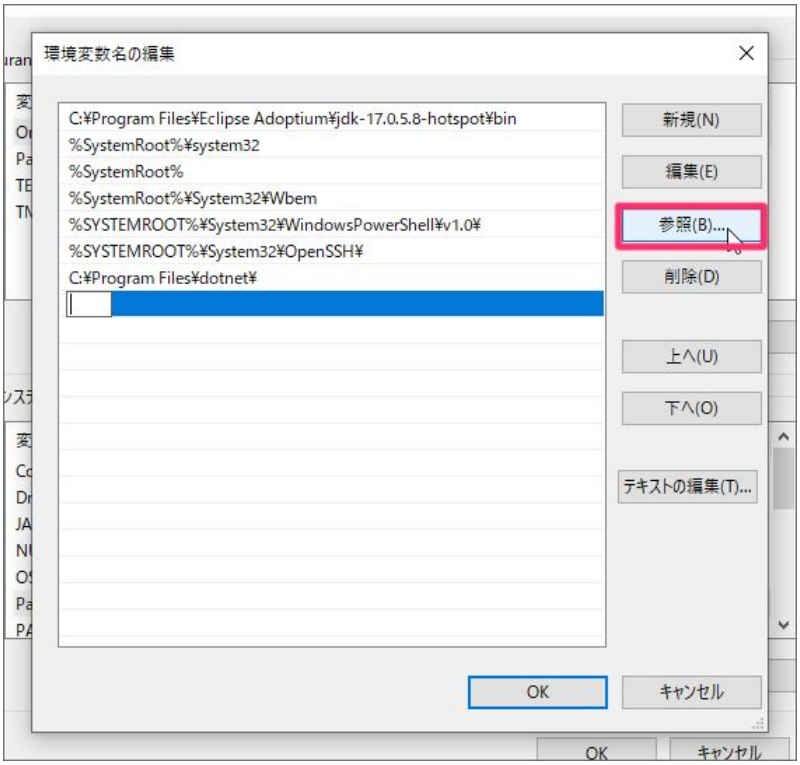


4)「環境変数名の編集」ダイアログが表示される。

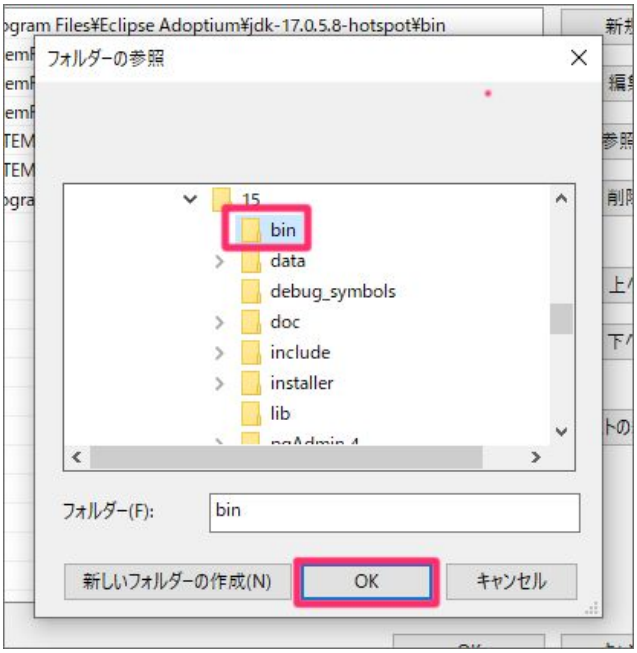
⇒ [新規(N)]ボタンをクリックする。



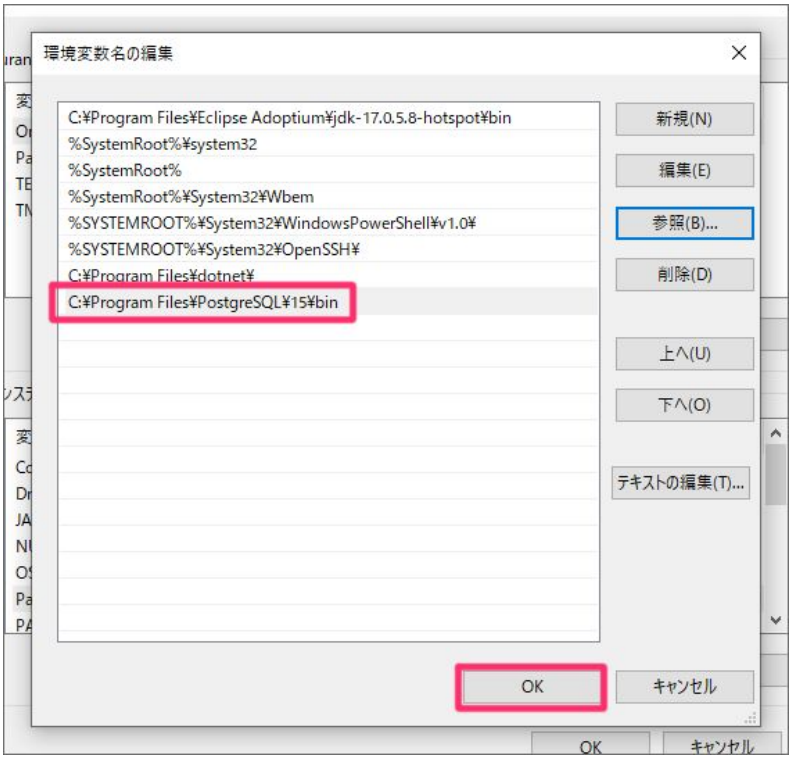
5)[参照(B)...]ボタンをクリックする。



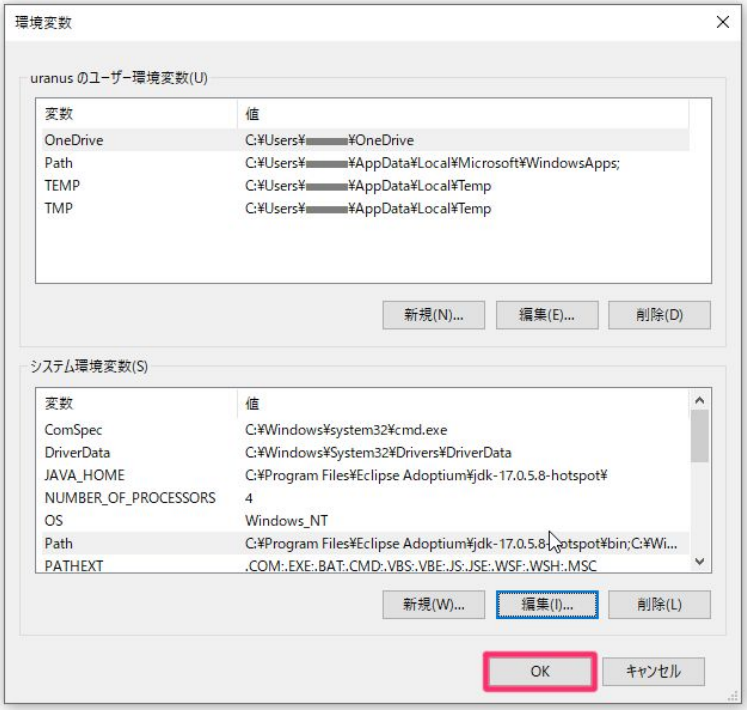
6)<POSTGRES_DIR>\binを選択 > [OK]ボタンをクリックする。



7)<POSTGRES_DIR>\binが追加されたことを確認 > [OK]ボタンをクリックする。



8)環境変数ダイアログに戻るので[OK]ボタンをクリックして閉じる。

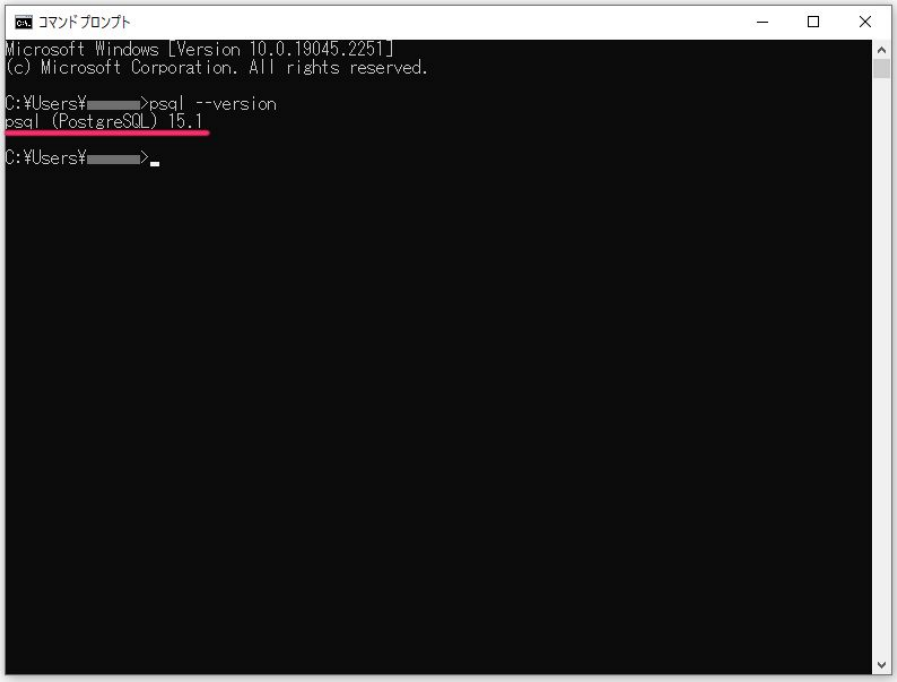


9)PostgreSQLが正常にインストールできたことを確認する。

コマンドプロンプトを開き、以下のコマンドを入力する

```
psql --version
```

⇒ Versionが表示されることを確認する。



⇒exitと入力するか、右上の[X]をクリックしてコマンドプロンプトを閉じる。

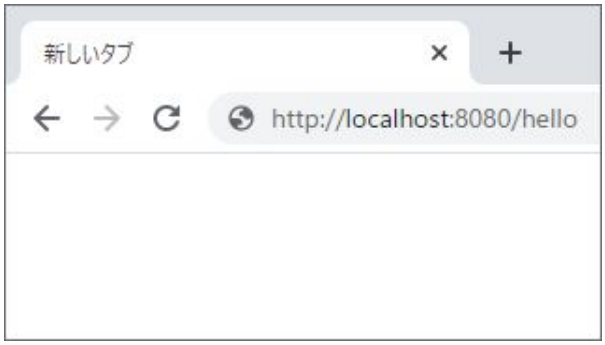
3. Spring BootでHello,world!

3.1 Hello,world!

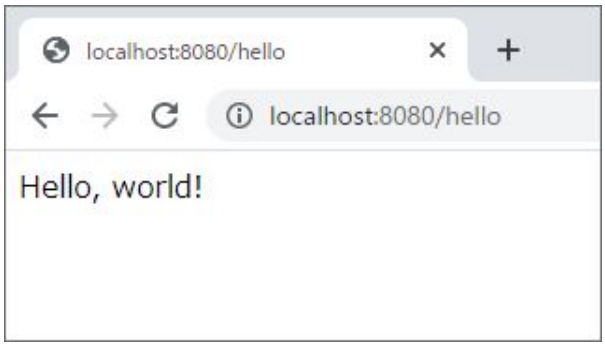
多くのプログラミング入門書では、"Hello, world!"といった文字列を表示する、いわゆる「Hello worldプログラム」から始めています。そこで本書もこれに習い、最初のプログラムはSpring Boot版Hello worldです。完成すると以下のようになります。

■実行例

1)ブラウザを起動しhttp://localhost:8080/helloを開く。

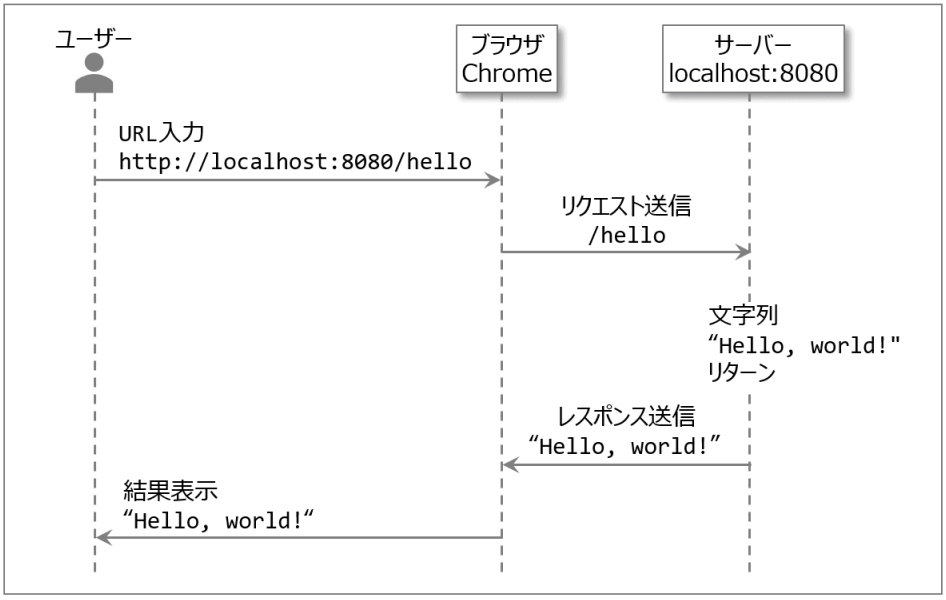


2)"Hello, world!"と表示される



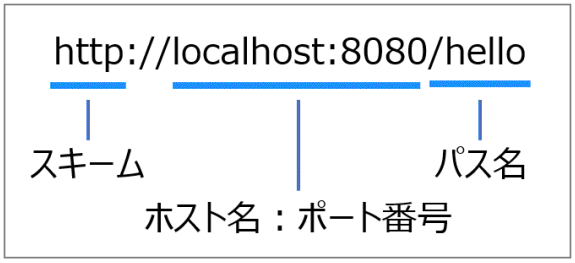
このようにとても単純なものです。ですが、内部ではWebアプリケーションの基礎である**リクエスト(Request)/レスポンス(Response)**を使っています。「Hello, world」では、これを学ぶことができます。

上記の操作をUML(Unified Modeling Language)のシーケンス図で表すと、次のようになります。



【図3-1】Hello, worldのシーケンス

リクエストは、サーバーに対する何らかの**「要求」**です。この要求は**URL**で表します。
先ほどブラウザへ入力したURLは、次のような形をしています。



これで「**localhost**(=自マシン)の**8080**番ポートへ接続し、**http**の規約に従って**/hello**というデータを取得せよ」という要求になります。

リクエストを受け取ったサーバー(=localhost)は、/helloというパス名を処理するプログラムを探し、そこに処理を依頼します。すると"Hello, world!"という文字列が返されるので、それをブラウザへ送信します。**レスポンス**は、この**「応答」**のことです。
そして「パス名」と、それに対応する「処理」をペアにして持っているのが、**「コントローラークラス」**と呼ばれるJavaで記述したプログラムです。

では実際にSTSを使ってHello, world!を表示させてみましょう。

プロジェクトを作成する

1)STSを起動する

2.2節でインストールしたSTSを起動する。

2)「ワークスペースとしてのディレクトリー選択」ダイアログが表示される。

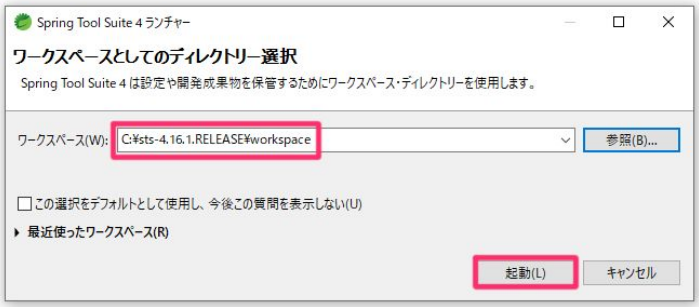
STSではJavaプログラムやHTMLファイル、設定情報など関連するリソースを**「プロジェクト」**という単位で管理します。

そしてプロジェクトを格納するフォルダを**「ワークスペース」**と言います。つまりワークスペースは、プロジェクトを保管するフォルダです。このダイアログは、そのフォルダを選択するものです。

デフォルトでよければ、[起動(L)]ボタンをクリックします。

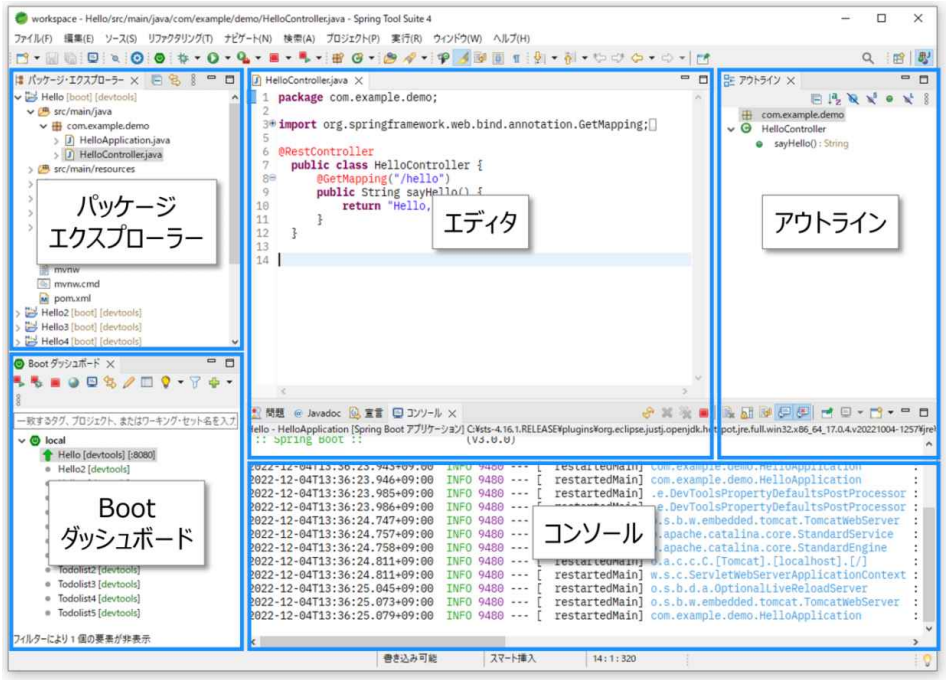
変更する場合は[ワークスペース(W)]にフォルダ名を入力してから[起動(L)]ボタンをクリックします。

本書ではワークスペースを以下のように**<STS_DIR>\workspace**とします。



3)STSの初期画面

STSの画面は、以下のような領域から構成されています。この構成を「Javaパースペクティブ」と言います。

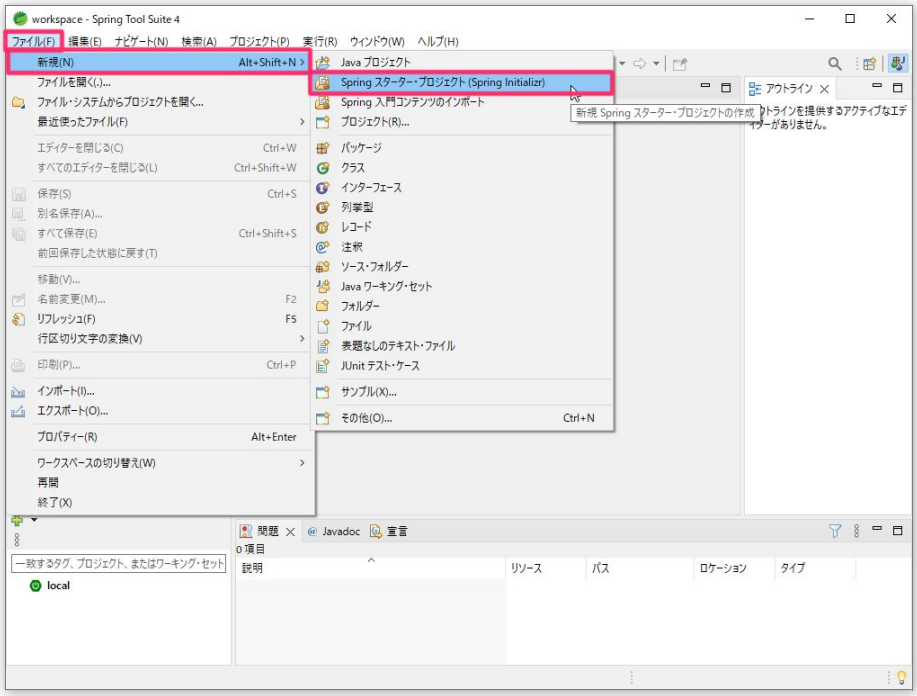


【表3-1】Javaパースペクティブを構成する領域

領域名	機能
パッケージ・エクスプローラー	プロジェクトのリソースを階層的に表示・操作する
Bootダッシュボード	プロジェクトに対応するサーバーを表示・操作する
エディタ	JavaプログラムやHTMLファイルなどを表示・編集する
コンソール	サーバーの実行状況を表示する
アウトライン	編集中のファイルのアウトラインを表示する

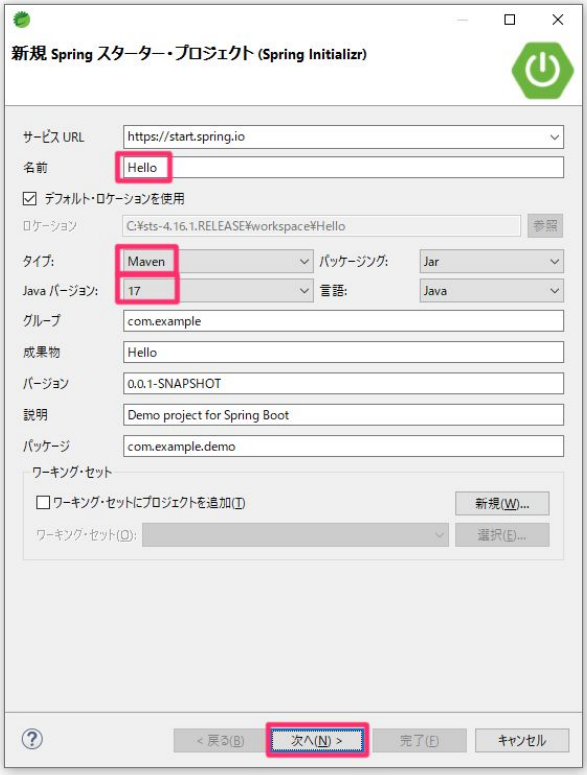
4)プロジェクトを作成する

STSのメニューから[ファイル(F)] > [新規(N)] > [Spring スターター・プロジェクト]を選択する。



「Spring スターター・プロジェクト」は「Spring Bootを使うのに必要な設定が済んでいるプロジェクトのひな型」です。この後ダイアログから各種情報を入力すると、その内容を反映したプロジェクトが自動的に作成されます。

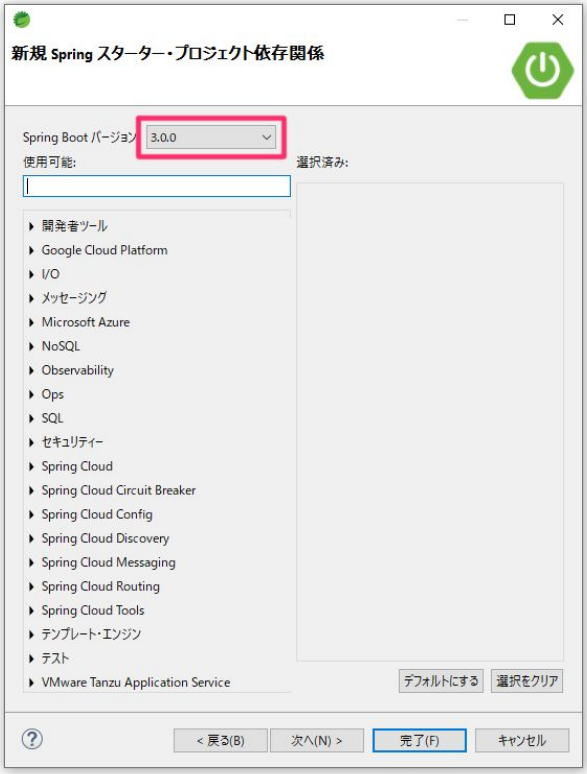
5)「新規Springスターター・プロジェクト」ダイアログが表示される。
[名前]に**"Hello"**と入力します。これが作成する**プロジェクトの名前**になります。
⇒ [名前]を入力すると、[ロケーション]、[成果物]が自動的に変更されます。
[タイプ]は **Maven** を選択してください。
また[Javaバージョン]が **17** になっていることを確認してください。
それ以外はデフォルトのままで大丈夫です。



[次へ(N)>]ボタンをクリックします。

> デフォルトでタイプは**Gradle**が選択されていると思います。このMavenやGradleはプロジェクトで使う**ビルドツール**のことです。興味がある方は調べてみてください。

6)依存関係を設定する



次に表示される「新規Springスターター・プロジェクト依存関係」では、このプロジェクトで使用するフレームワークやライブラリを選択します。

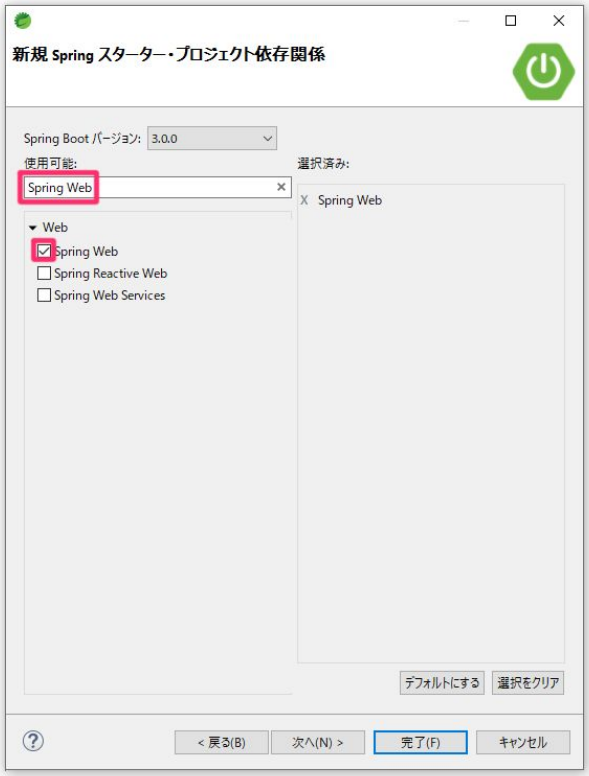
まずフレームワークですが[Spring Bootバージョン]で**3.0.0**(あるいはそれ以降)を選択してください。2.7.Xなどを選択すると、本書のプログラムで文法エラーになるところがあります。

またHelloプロジェクトでは、以下のライブラリを使用します。
[1]Spring Web
[2]Spring Boot DevTools

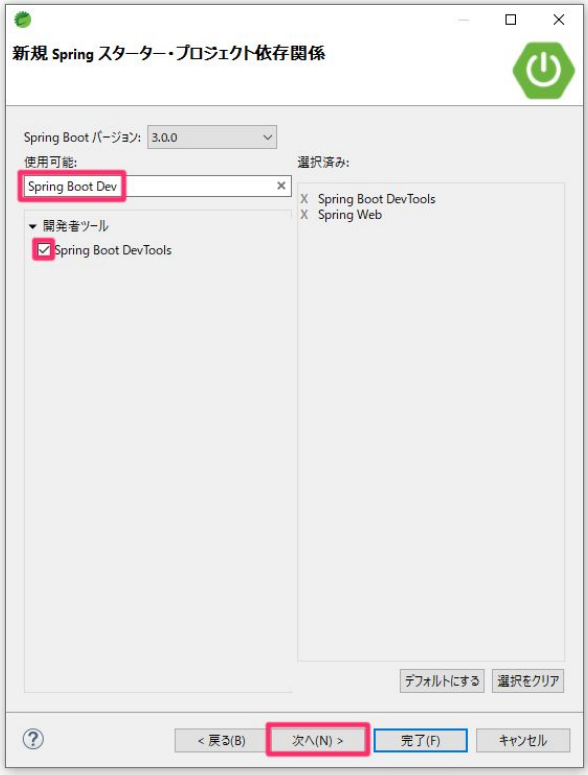
[1]はSpring BootでWebアプリケーションを作成するのに必須です。[2]はオプションですが、作業効率化に役立つので、合わせて選択することをお勧めします(具体的な効用は後述します)。

選択の操作は、まず[使用可能]に**"Spring Web"**と入力します。するとこのキーワードを含むものが表示されるので、その中から「**Spring Web**」をチェックします。これで

右側の「選択済み」に「Spring Web」が表示されます。



続けて今度は[使用可能]に"**Spring Boot Dev**"と入力します。「**Spring Boot DevTools**」が表示されるので、これもチェックします。

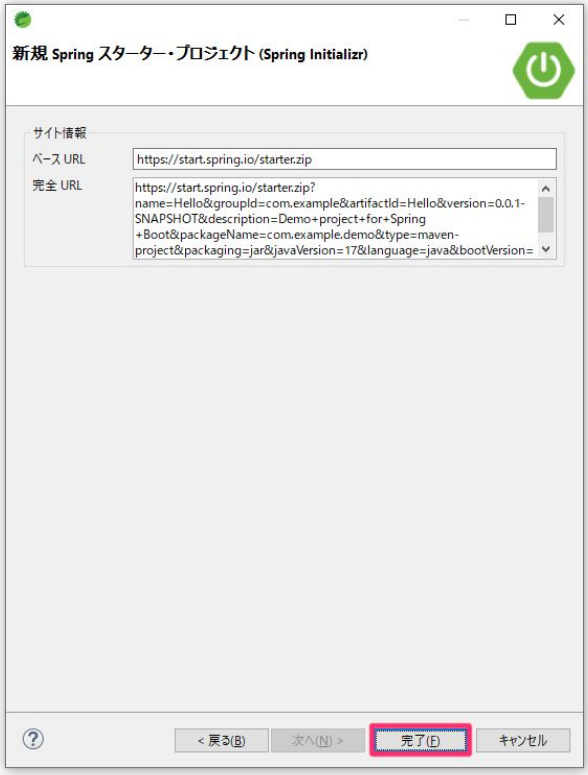


なおこの２つは本書のすべてのプロジェクトで使います。選択済みの状態で[**デフォルトにする**]ボタンをクリックすると、次のプロジェクトから選択する手間が省けます。

[次へ(N)>]ボタンをクリックします。

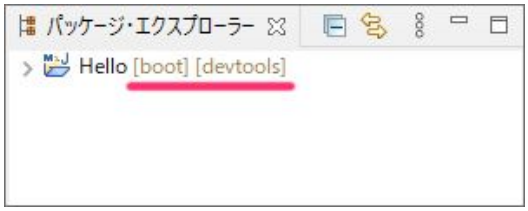
7)プロジェクトを作成する

[完了(F)]ボタンをクリックします。これでHelloプロジェクトが作成されます。



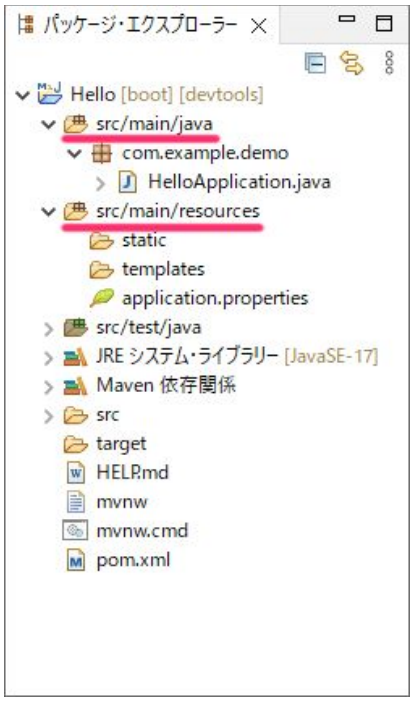
STSで初めてプロジェクトを作成するときは、インターネット経由で多くのリソースをダウンロードするため、少々時間がかかります(回線速度が遅いと10分以上かかることもあります)。2回目以降は、差分だけなので早くなります。インターネットに接続していないと、プロジェクトを作れないので注意してください。

プロジェクトが作成されると、パッケージ・エクスプローラーに表示されます。
⇒Helloの右側に**[boot][devtools]**と表示されるまで待ちます。



8)作成されたプロジェクト内容

パッケージ・エクスプローラーのHello左にある[>]をクリックしていくと、プロジェクトの内容を表示できます。



src/main/java下には、これからJavaクラスを定義していきます。
なお**HelloApplication.java**はプロジェクト作成時、自動的に作成されたクラスで、このプロジェクトを実行するときのエントリーポイントです(Javaプログラムの`main()`のようなイメージです)。ファイル名は「プロジェクト名」+ `"Application.java"`という意味です。

src/main/resources下のフォルダ、ファイルの用途は以下のようになっています。

【表3-2】src/main/resources下のフォルダおよびファイル

フォルダ/ファイル	用途
`static`	CSSファイルなど、内容が変化しないファイルを格納するフォルダ
`templates`	処理結果画面など、内容が動的に変化するファイルを格納するフォルダ
`application.properties`	DB接続情報など、プロジェクト全般に関わる設定情報を記述するファイル

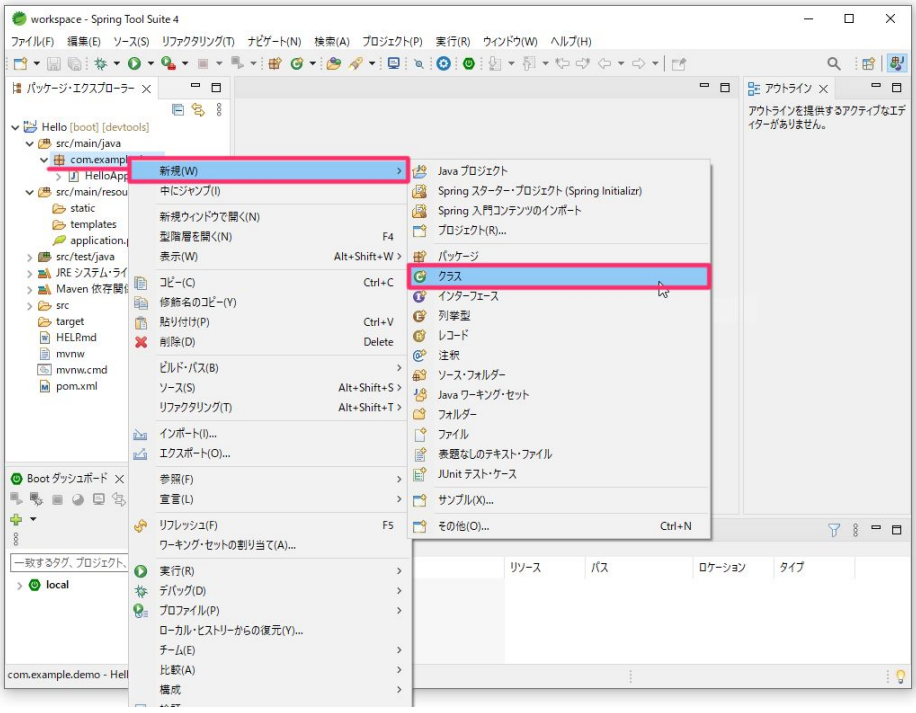
詳細は後述します。

コントローラークラスを作成する

プロジェクトができたので、ここに「URLのパス名」と「対応する処理」をペアで持つ**「コントローラークラス」**を追加します。名称は自由ですが、このクラスの役割は"Hello, world!"という文字列を返すことなので、「**HelloController**」とします。このようにコントローラークラスの名称は、～Controllerとするのが一般的です。

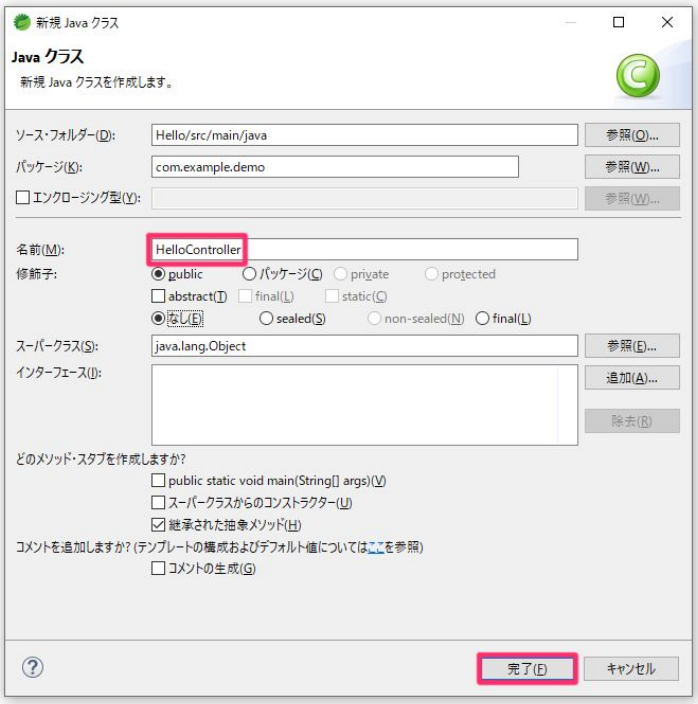
作成手順は以下のようになります。

1)パッケージ・エクスプローラーで**com.example.demo**を右クリック ＞ [新規(W)] ＞ [クラス]を選択する。

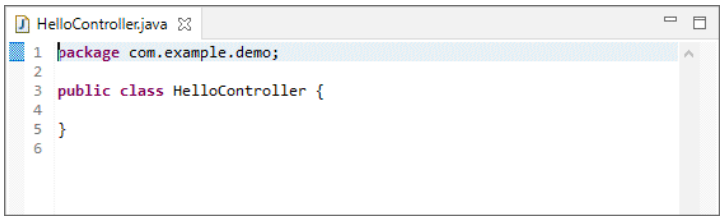


2)「新規Javaクラス」ダイアログが表示される。

[名前(M)]に"**HelloController**"と入力 ＞ [完了(F)]ボタンをクリックする。



3)作成されたHelloController.javaを次のように編集する(保存は[CTRL]+S)。



↓

【リスト3-1】com.example.demo.HelloController.java

```
package com.example.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController                         //  ①
public class HelloController {
    @GetMapping("/hello")               //  ②
    public String sayHello() {
        return "Hello, world!";
    }
```

```
}
```

ポイントは@で始まる**アノテーション(注釈)**という付加情報です。
アノテーションは直後にあるものに対して、いろいろな情報を付与します。

ここでは２つのアノテーションを使っています。

①@RestControllerアノテーション

「コントローラークラスのメソッドで処理した結果を、そのままレスポンスとしてブラウザへ送信する」ことを表すアノテーションです。本来はJSONやXMLなどを返す「RESTインターフェース」で使うものですが、「テキストを返す」機能を流用できるので、このアノテーションを利用します(画面用のアノテーションは後述します)。

②@GetMappingアノテーション

「GETリクエストに対応するメソッド」であることを表すアノテーションです。
「GETリクエスト」は、サーバーへリクエストを送る方法の１つです。詳しくは3.2節で説明しますが、
ブラウザのアドレス欄に

```
http://localhost:8080/hello
```

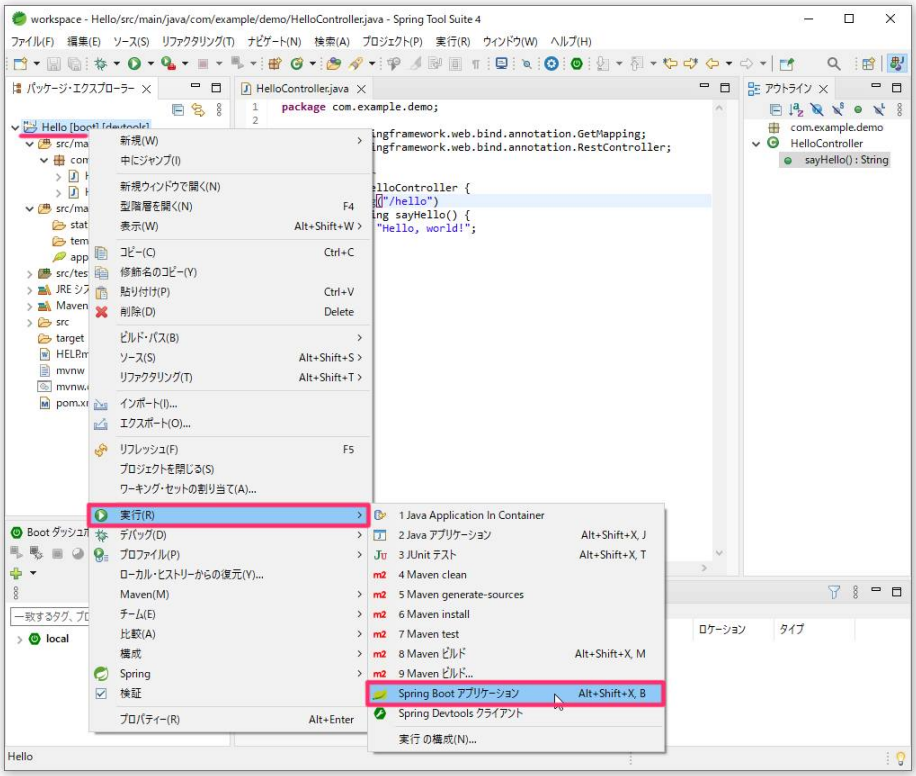
といったURLを入力すると、サーバーには「GETリクエスト」として送信されます。
@GetMappingは、このGETリクエストを処理するメソッドであることを表しています。

@GetMappingの引数には処理対象とする「**URLのパス名**」を書きます。
@GetMapping("/hello")とすると、GETリクエストで/helloが送られてきたら、直後にあるsayHello()が**自動的に呼び出される**ようにしてくれます。これが先に説明した「URLパス名に対応する処理」になります。このようなメソッドを**「ハンドラーメソッド」**と言います。

プロジェクトの実行

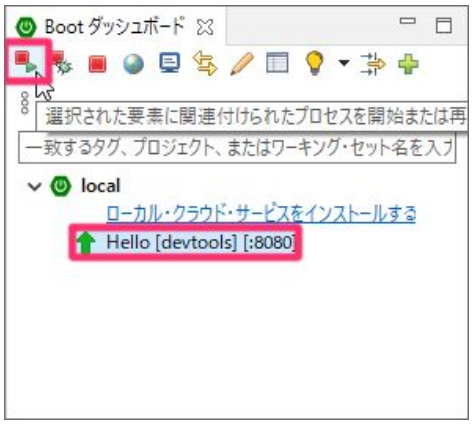
では実行してみましょう。
まず以下のような方法で、プロジェクトを実行します。

1-a)パッケージ・エクスプローラーの**Helloプロジェクト**を右クリック > [実行(R)] > [Spring Boot アプリケーション]を選択する。



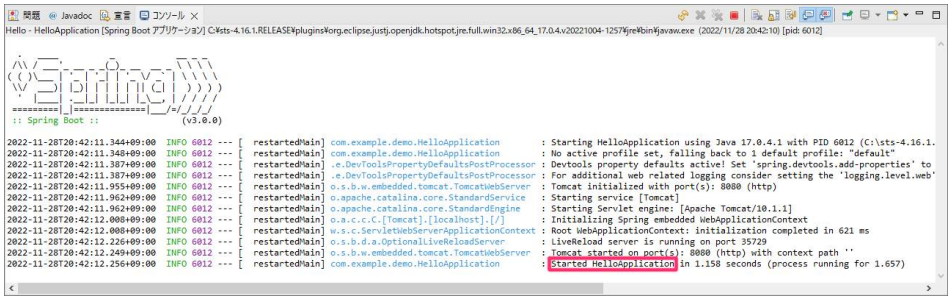
または

1-b)Bootダッシュボード > Localの[>]をクリック > Helloを選択 > [(再)開始]ボタンをクリックする。

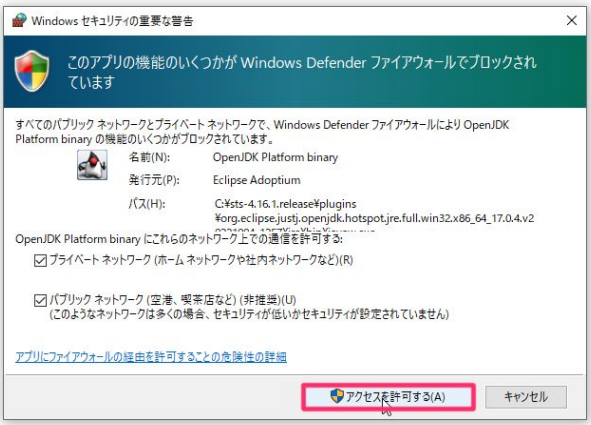


2)コンソールにメッセージが出力され始めます。

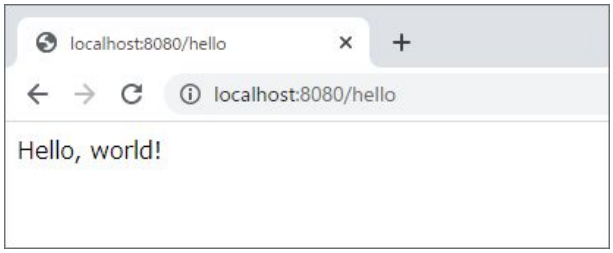
⇒”Started HelloApplication ...“が出力されるまで待ちます。



3)初めてSTSでプロジェクトを実行するとき、以下のようなメッセージが表示されることがあります。この場合は[アクセスを許可する(A)]ボタンをクリックしてください。



4)Helloプロジェクトが起動したら、ブラウザのアドレス欄に
http://localhost:8080/helloと入力します。
⇒"Hello, world!"と表示されればOKです。



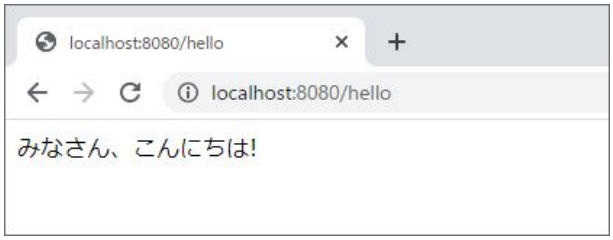
5)成功したらSTSに戻って、HelloController.javaのreturn文を以下のように変更→保存してください。

⇒再びコンソールにメッセージが出力され始めるので、止まるまで待ちます。

```
return "Hello, world!";
    ↓
return "みなさん，こんにちは!";
```

6)ブラウザの再読み込みボタンを押すか、[CTRL]+[R]または[F5]を押して再読み込みします。

⇒表示内容が変わればOKです。



このようにプログラムを変更→保存すると、再デプロイ(deploy)やサーバーを再起動することなく、すぐ反映されます。これが依存関係で選択したSpring Boot DevToolsの機能です。変更した結果を素早く確認できるので、効率良く作業を行えます。

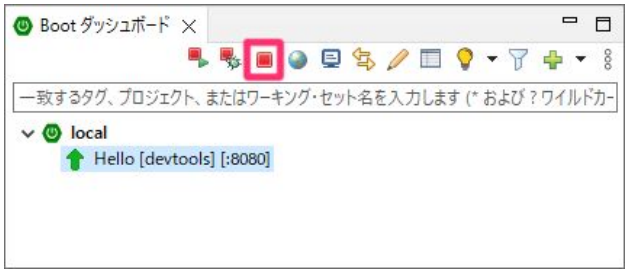
プロジェクトの停止

プロジェクトを停止するときは、以下のような方法で行います。

1-a)パッケージ・エクスプローラーでHelloプロジェクトを選択 > ツールバーの[停止]ボタンをクリックする。

または

1-b)Bootダッシュボードで`Hello`を選択 > [停止]ボタンをクリックする。



3.2 GETリクエスト/POSTリクエスト

一般にWebアプリケーションでは、ブラウザから処理して欲しいデータなどをリクエストに乗せてサーバーへ送ります。リクエストには何種類かありますが、本書では**「GETリクエスト」「POSTリクエスト」**を使います。

この２つはリクエストをどこから送るか？で使い分けます。

【表3-3】GETリクエストとPOSTリクエスト

リクエスト	リクエスト送信元
GETリクエスト	・ブラウザのアドレス欄 ・リンク(a要素) ・method属性にgetを指定したform要素(<form>～</form>)
POSTリクエスト	・method属性にpostを指定したform要素(<form>～</form>)

一般的にGETリクエストはサーバーからのデータ取得、POSTリクエストはデータ登録で使われます。

以下GETリクエスト、POSTリクエストでサーバーへデータを送る方法を説明します。

> この「GETリクエスト」「POSTリクエスト」は、「HTTPのGETメソッドを指定したリクエスト」、同じく「POSTメソッドを指定したリクエスト」のことです。メソッドには他にもPUT，DELETE，HEAD，OPTIONS，TRACEなどがあります。興味がある方は調べてみてください。

3.3 GETリクエストの処理

GETリクエストでは、サーバーへ送るデータをURLに付加します。その方法には次の２つがあります。

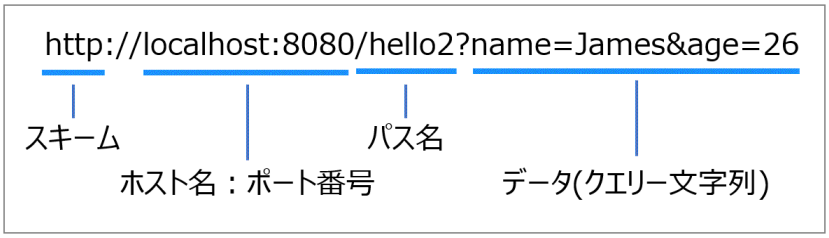
①クエリ文字列

②URIパス変数

3.3.1 クエリ文字列からのデータ取得

インターネットをブラウズしていると、**?**を含むURLがブラウザのアドレス欄に表示されることがあります。

例.



この?に続くのがデータです。これを**クエリ文字列**(Query String)と言います。**「パラメータ名=パラメータ値」**という形式で、複数あるときは**&**で繋げます。上記のURLなら、下表のようなパラメータがサーバーへ送信されます。

【表3-4】クエリ文字列name=James&age=26の意味

パラメータ名	パラメータ値	備考
name	James	
age	26	文字列として送られる

ではプロジェクトを追加し、新しいHello,worldプログラムを作成します。今度はブラウザから名前を渡せるようにします。

URLパスは**/hello2**とし、名前は**name=James**のように、クエリ文字列で指定します。

これをブラウザから入力すると、以下のようにnameの値と合わせて、テキストが表示されます。

■http://localhost:8080/hello2?name=Jamesの実行結果

作成するプロジェクトの仕様

プロジェクトの作成手順は、前節のHelloプロジェクトと同様です。ここにはキーになる項目だけ記載します。

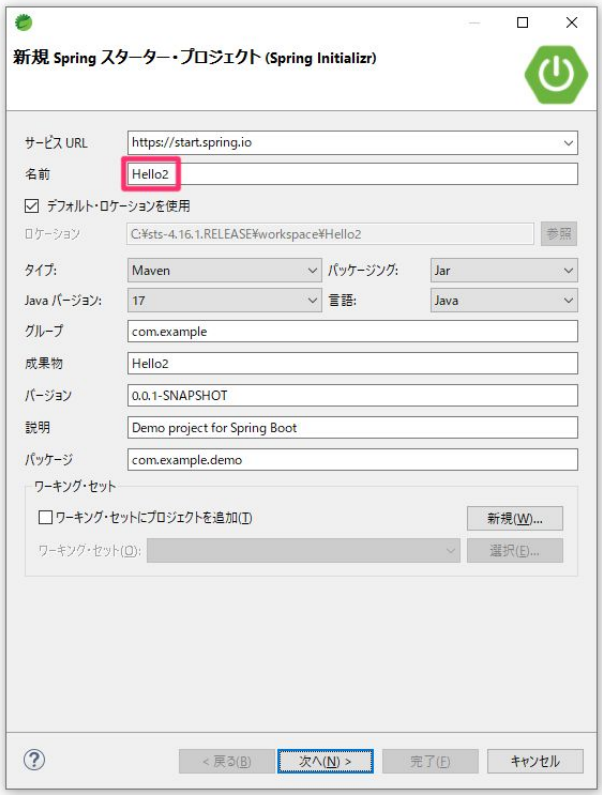
プロジェクト名	Hello2
依存関係	Spring Web, Spring Boot DevTools
コントローラークラス	Hello2Controller

1)プロジェクトの作成

STSのメニューから[ファイル(F)] > [新規(N)] > [Spring スターター・プロジェクト]を選択。

「新規Springスターター・プロジェクト」で[名前]に"**Hello2**"を入力。

[タイプ]は**Maven**, [Javaバージョン]は**17**とします。

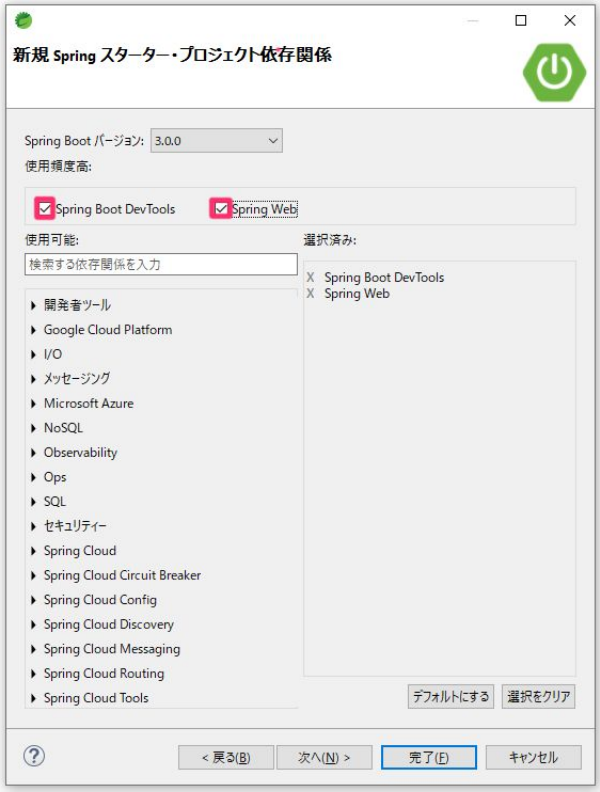


2)依存関係の指定

Spring Web, Spring Boot DevToolsを選択。

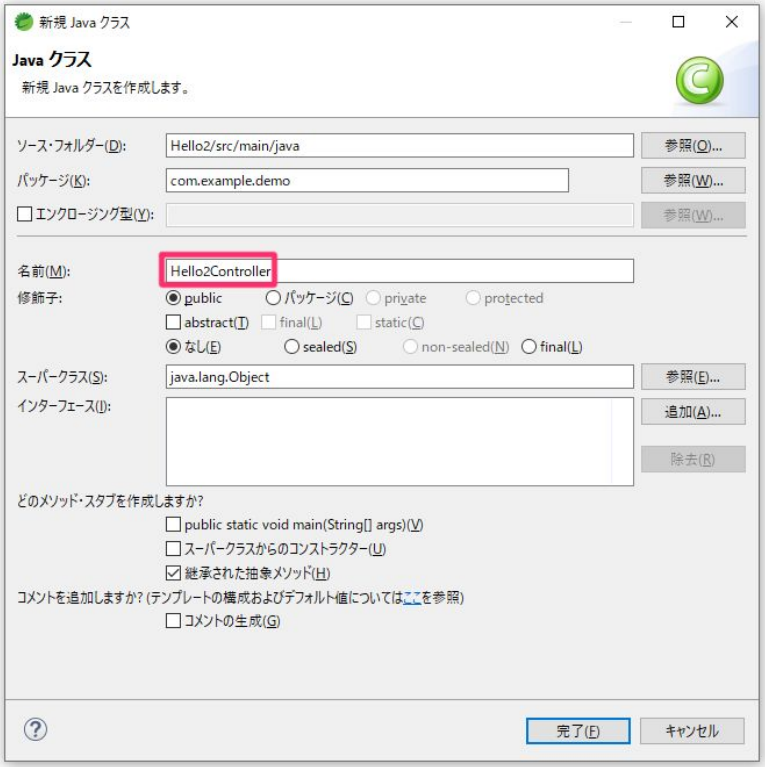
⇒Helloプロジェクトで使用したものが[使用頻度高]に表示されていれば、それをチェックします。

⇒表示されていない場合は、Helloプロジェクトと同じ手順で選択してください。



3)コントローラークラスの作成

パッケージ・エクスプローラーで**Hello2**の**com.example.demo**を右クリック > [新規(W)] > [クラス]を選択する。[名前(M)]に**"Hello2Controller"**と入力する。



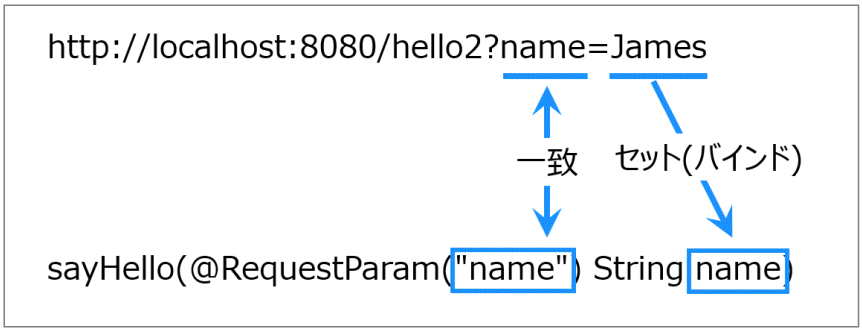
Hello2Controllerを、以下のように編集します。

【リスト3-2】com.example.demo.Hello2Controller.java

```
package com.example.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class Hello2Controller {
    @GetMapping("/hello2")
    public String sayHello(@RequestParam("name") String name) {
        return "Hello, world! " + "こんにちは " + name + "さん!";
    }
}
```

基本的な構造はHelloControllerと同じです。違いはsayHello()に引数があり、**@RequestParamアノテーション**が付与されていることです。

@RequestParamは引数で指定されたパラメータの値を、メソッド実行前にクエリ文字列から取得し、メソッド引数へセット(バインド)します。



このようにsayHello()は、引数nameに"James"が設定された状態で始まります。これを使ってブラウザに返信する文字列(=レスポンス)を作成します。

@RequestParamの引数は、クエリ文字列のパラメータ名と一致させます。

一方、バインドされる引数名は自由です。たとえば次のようにすると、引数onamaeに"James"がセットされます。

```
public String sayHello(@RequestParam("name") String onamae)   //   OK
```

しかし次の場合は、クエリ文字列にonamaeという名前のパラメータが無いため、実行時エラーが発生します。

```
public String sayHello(@RequestParam("onamae") String name)   //   NG
```

パラメータが複数あるときは、その分@RequestParamを付与したメソッド引数を記述します。

```
http://localhost:8080/hello2?name=James&age=26
↓
public String sayHello(@RequestParam("name") String name, @RequestParam("age") int age)
```

メソッドの引数ageはint型なので、自動的にintに変換されてバインドされます。

なお@RequestParamの名称と、バインドされる引数名が同じなら、@RequestParamは省略できます。よって以下のようにすることもできます。

```
public String sayHello(@RequestParam("name") String name, @RequestParam("age") int age)
↓
public String sayHello(String name, int age)
```

3.3.2 URLパスからのデータ取得

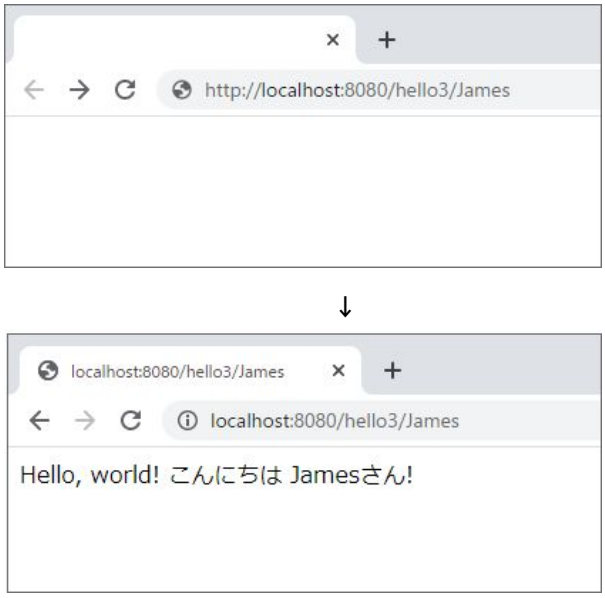
GETリクエストでデータを送るもう１つの方法は、URLパスにその値を含めてしまうものです。

たとえばURLパス**/hello3**へ**“James”**というデータを送るとき、下記のようにすることもできます。

http://localhost:8080/hello3/James

次のプロジェクトでは、この値を取り出して表示します。実行イメージは以下のようになります。

■http://localhost:8080/hello3/Jamesの実行結果



作成するプロジェクトの仕様

プロジェクト名	Hello3
依存関係	Spring Web, Spring Boot DevTools
コントローラークラス	Hello3Controller

【リスト3-3】com.example.demo.Hello3Controller.java

```java
package com.example.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class Hello3Controller {
    @GetMapping("/hello3/{name}")
    public String sayHello(@PathVariable("name") String name) {
      return "Hello, world! " + "こんにちは " + name + "さん! ";
    }
}
```

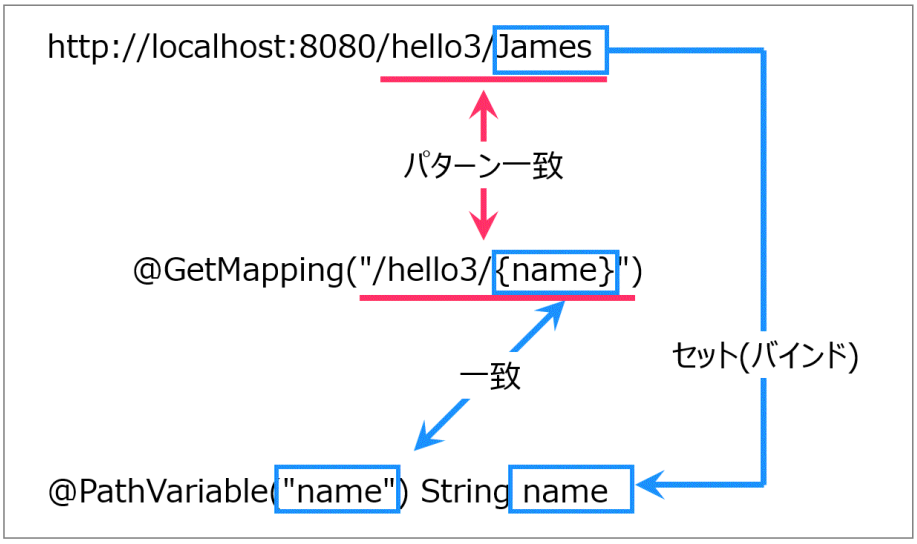
前節のHello2Controllerと、以下の点が異なります。

1)@GetMappingアノテーション

引数に**{}**で囲まれた部分があります。これはURLパスから値として取り出す部分を指定するもので**「URIテンプレート変数」**と言います。この場合、URLパスが/hello3/Jamesなら、nameという名前で"James"を取り出せます。

2)**@PathVariableアノテーション**

@RequestParamが@PathVariableに変わっています。このアノテーションは、1)のURIテンプレート変数の値をメソッド引数にセット(バインド)します。



なお@PathVariableの名前は、URIテンプレート変数名と一致させる必要があります。

```
@GetMapping("/hello3/{onamae}")
public String sayHello(@PathVariable("onamae") String name) {   //  OK
```

```
@GetMapping("/hello3/{name}")
public String sayHello(@PathVariable("onamae") String name) {   //  NG
```

複数の値を埋め込むときは、以下のようにします。

```
http://localhost:8080/hello3/James/26
↓
@GetMapping("/hello3/{name}/{age}")
public String sayHello(@PathVariable("name") String name, @PathVariable("age") int
age)
```

3.4 POSTリクエストの処理

次にPOSTリクエストですが、POSTリクエストは、会員登録画面など**HTMLのform要素**で作成した画面に入力されたデータを、サーバーへ送るのに使います。本節ではformに入力された名前を取得し、Hello, world!と合わせて出力します。実行イメージは以下のようになります。

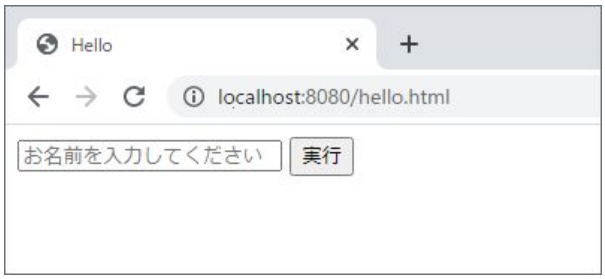
■実行例

1)入力画面(hello.html)を要求する(GETリクエスト)：

http://localhost:8080/hello.html

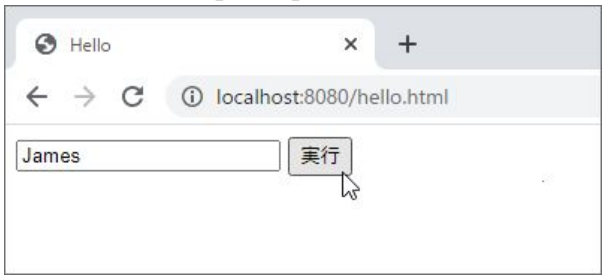
↓

2)入力画面が表示される(レスポンス)



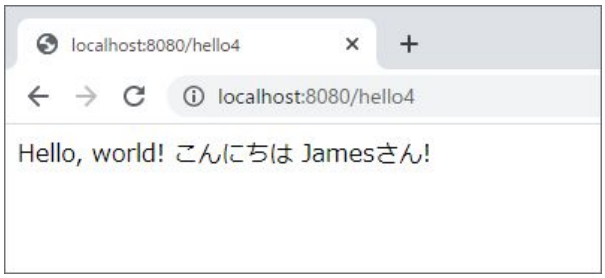
↓

3)名前を入力して[実行]ボタンをクリックする(POSTリクエスト)

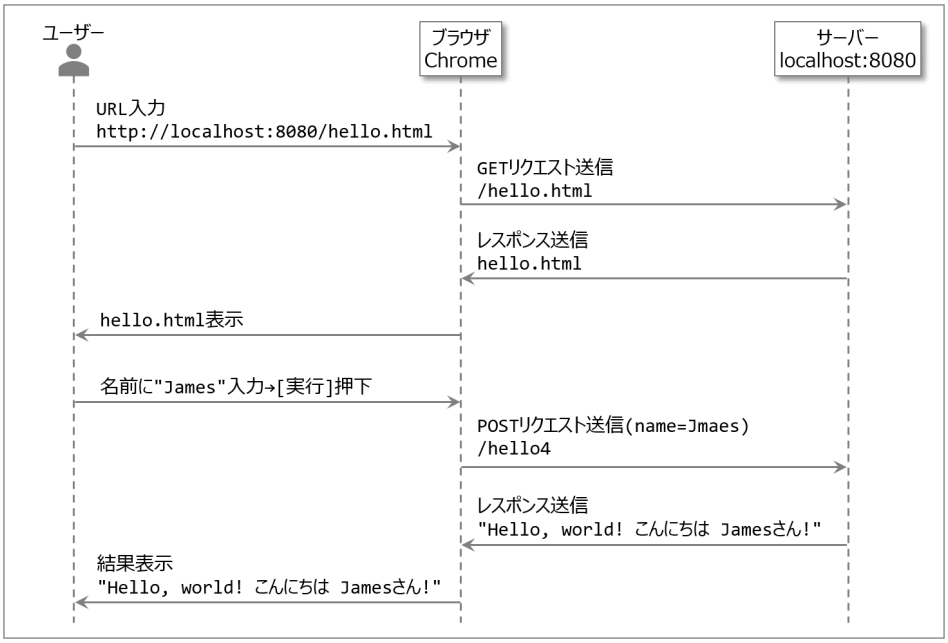


↓

4)結果表示(レスポンス)



この操作ではブラウザ～サーバー間をリクエスト/レスポンスが2往復しています。



【図3-2】POSTリクエスト版Hello，worldのシーケンス

最初の画面要求は、ブラウザのアドレス欄にURL(http://localhost:8080/hello.html)を入力するのでGETリクエストです。このリクエストのように(URLパスでなく)ファイル名を指定すると、そのファイルがレスポンスとして返されます。

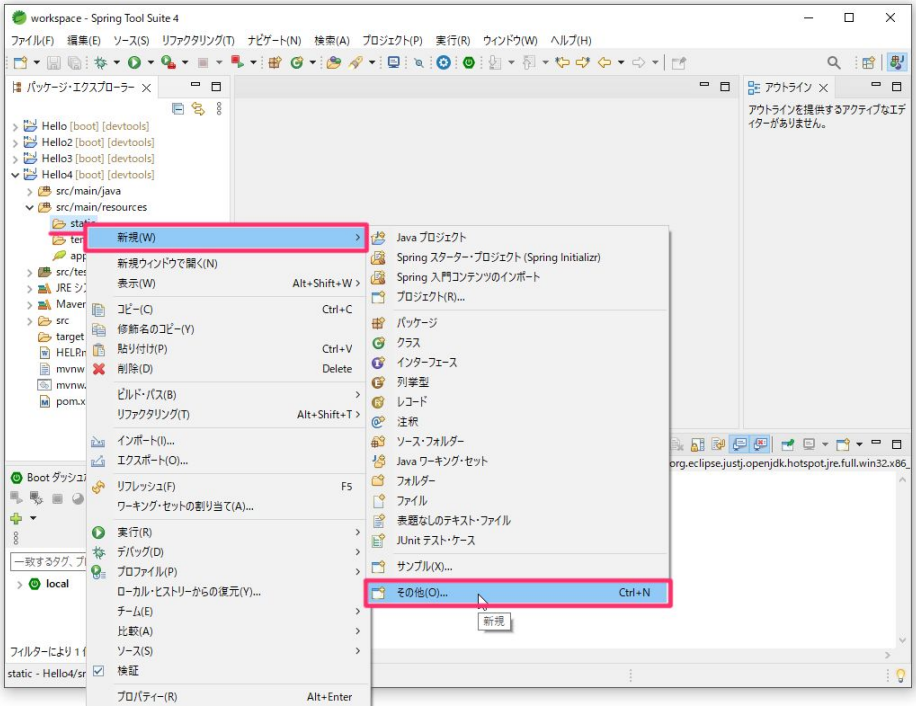
hello.htmlで[実行]ボタンをクリックすると、POSTリクエストが送信されます。入力されたデータは、クエリ文字列の形式(name=James)でリクエストの「メッセージボディー」という部分に格納され、サーバーへ送られます。メッセージボディーについては割愛しますが、POSTリクエストもGETリクエスト同様、**パラメータ名(ここではname)がキーになる**ことを覚えておいてください。

作成するプロジェクトの仕様

プロジェクト名	Hello4
依存関係	Spring Web, Spring Boot DevTools
コントローラークラス	Hello4Controller
作成ファイル	src/main/resources/static/hello.html(入力画面)

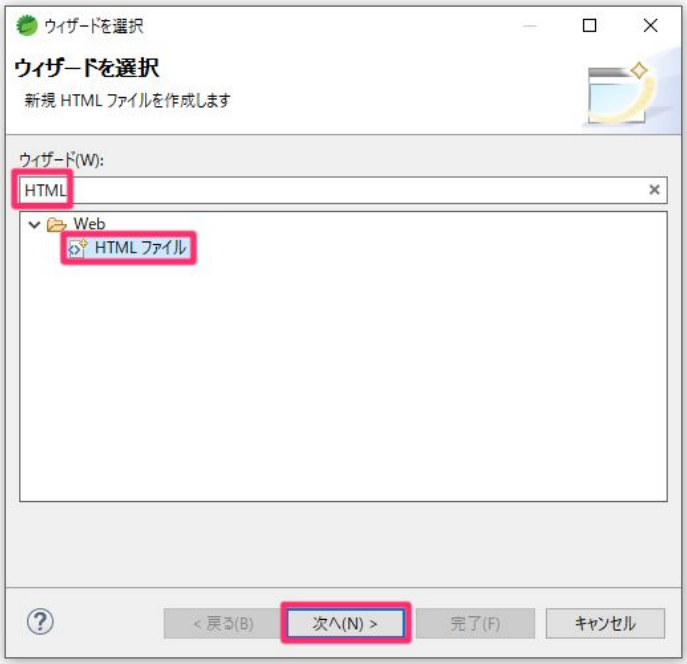
hello.htmlファイルは、以下のように作成します。

1)パッケージ・エクスプローラーの**Hello4**を展開し**src/main/resources**下の**static**を右クリック > [新規(W)] > [その他(O)]を選択。

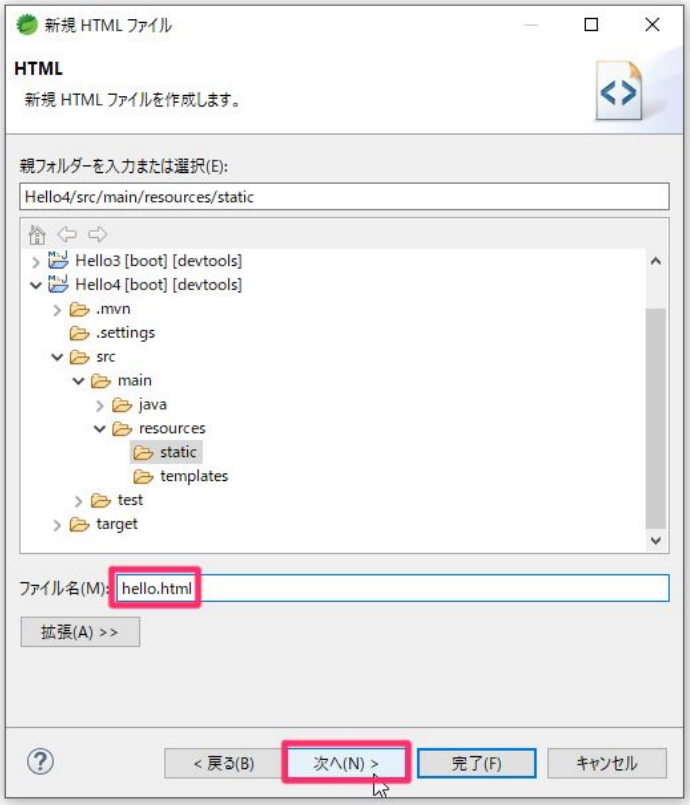


1)「新規」ダイアログの[ウィザード(W)]に"**HTML**"と入力。表示された[HTMLファイル]を選択 > [次へ(N)>]ボタンをクリックする。

※**Eclipse Web 開発者ツール**がインストールされていないと、ここで「HTMLファイル」が表示されません。その場合は「2.2.3 Web開発用プラグインのインストール」の操作を行ってから実行してください。



3)「新規HTMLファイル」ダイアログの[ファイル名(M)]に"**hello.html**"と入力 > [次へ(N)>]ボタンをクリックする。



4)[**新規 HTMLファイル(5)**]が選択されていることを確認して、[完了(F)]ボタンをクリックする。

5)hello.htmlが作成される。

作成したhello.htmlは、以下のように編集します。

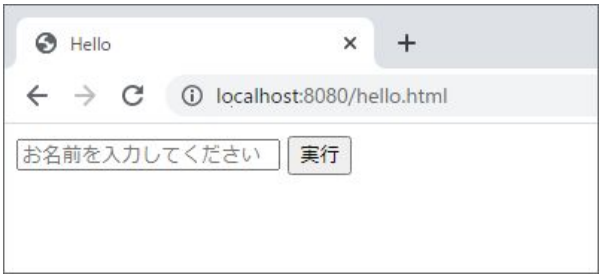
【リスト3-4】src/main/resources/static/hello.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Hello</title>
</head>
<body>
  <form action="/hello4" method="post">
```

```
    <input type="text" name="name" placeholder="お名前を入力してください">
    <input type="submit" value="実行">
  </form>
</body>
</html>
```

hello.htmlを作成した**staticフォルダ**は、静的(static)なファイル、言い換えると実行中に内容が変化しないファイルを格納する場所です。格納されているものと同名のファイルがリクエストされると、Spring Bootがその内容をブラウザへ自動的に返信します(コントローラークラスを経由しない)。

実行例と見比べると、最初の**input要素**は名前入力欄ということがわかると思います。



このようにinput要素は、ブラウザにデータを入力するための部品(**フォーム部品**)です。どのような形状にするかは**type属性**で決めます(詳細は次章で説明します)。

上記のようにtype="text"なら、テキスト入力フィールド(1行)になります。

name属性は、フォーム部品の名前であり、サーバーへ送信する際のパラメータ名に使われます。

また**placeholder属性**は、初期表示する文字列です(何か入力されると消える)。

もう１つのinput要素はtype="**submit**"です。これもブラウザと見比べると[実行]ボタンということがわかると思います(**value属性**の値がボタンのキャプション)。

submitボタンはクリックされたとき、**自分が属している**form要素に入力された内容を、クエリ文字列形式(パラメータ名=パラメータ値)にしてサーバーへ送信します。

この[実行]ボタンの場合、form要素の開始タグは、次のようになっています。

```
  <form action="/hello4" method="post">
```

action属性はデータの送信先、**method属性**は送信方法です。[実行]ボタンがクリックされたら、入力内容を/hello4へPOSTメソッドで送ります。データはクエリ文字列形式(name=Jmaes)になっていると思ってください。

なおmethod属性をgetにすれば、formからもGETリクエストを送れますが、あまり使う機会は無いと思います。

これを受け取るコントローラーは次のようにします。

【リスト3-5】com.example.demo.Hello4Controller.java

```
package com.example.demo;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class Hello4Controller {
    @PostMapping("/hello4")  //  ①
    public String sayHello(@RequestParam("name") String name) {  //  ②
        return "Hello, world! " + "こんにちは " + name + "さん!";
    }
}
```

ハンドラーメソッドに付与されている**@PostMappingアノテーション**は「POSTリクエストを処理するメソッド」ということを表しています。(①)。

フォーム部品に入力された値は、GETリクエストと同じくクエリ文字列形式(パラメータ名=パラメータ値)のため、@RequestParamで受け取ります(②)。

これでPOSTリクエストによって送られたフォームの入力値を取得できるようになります。

本章ではいろいろな方法で“Hello, world!”を表示させました。ここで説明した内容は、このあとも形を変えて頻繁に現れます。必要に応じて読み直してください。

参考：STSにプログラムを入力するときのヒント

STSでソースコードを入力するとき、import文を後回しにすると楽ができます。

たとえば以下のリストは、本章最初のものですが、このimport文を手打ちするのは少々骨が折れます(本書の終盤になるとimport文が10行以上になります)。

【リスト3-1】com.example.demo.HelloController.java(再掲)

```
package com.example.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, world!";
    }
}
```

こういった場合、import文は入力せず、あとからSTSの**クイックフィックス(Quick Fix)**機能で挿入すると、キータッチ数を減らせます。

以下、上記リストの入力を例に説明します。

1)import文を入力せず、"@RestController"を入力して改行すると、文法エラーを表す赤波線が引かれる。

⇒ 気にしないで次の行まで入力する。

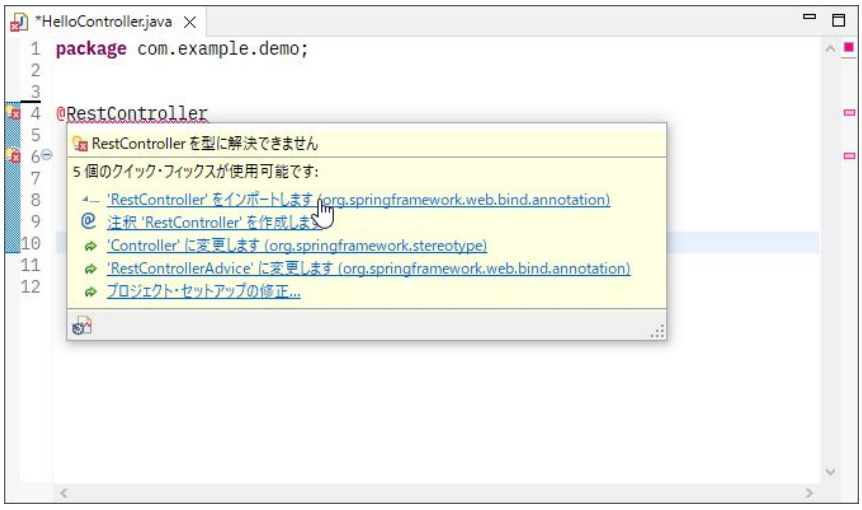
2)次の行まで入力したところ

3)@RestControllerにマウスカーソルを合わせる

⇒ポップアップウィンドウが表示される。

⇒これがクイックフィックス(Quick Fix)。文法エラーに対する解決策が表示されている。

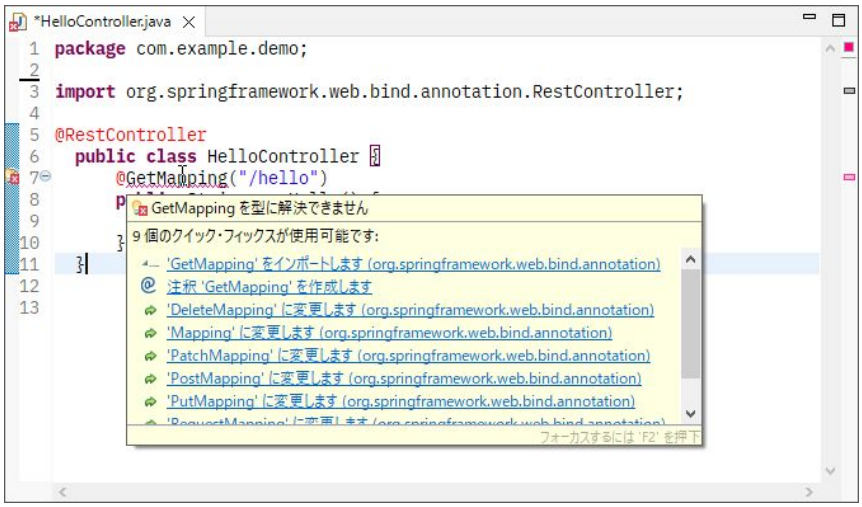
4)この中の「**'RestController'をインポートします**」のリンクをクリックする。



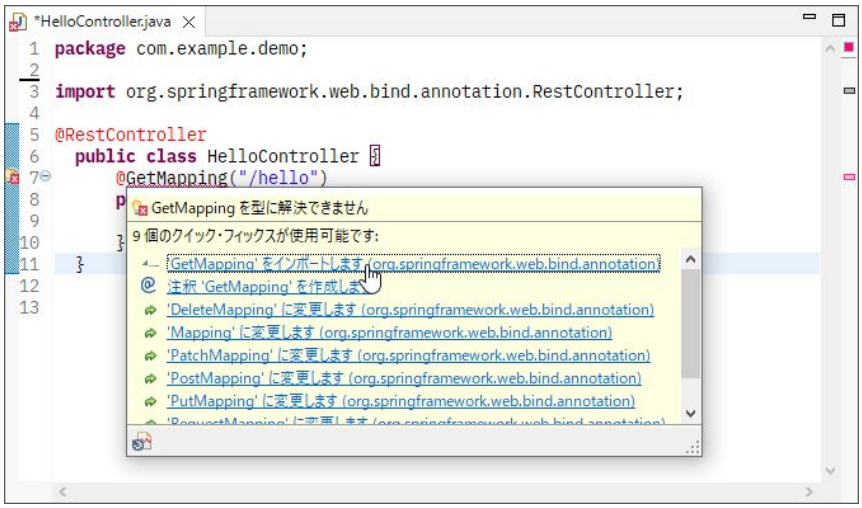
5)org.springframework.web.bind.annotation.RestControllerのimport文が挿入される。

⇒@RestControllerの赤波線も消える

6)@GetMappingも同じようマウスカーソルを合わせる⇒ポップアップウィンドウが表示される。

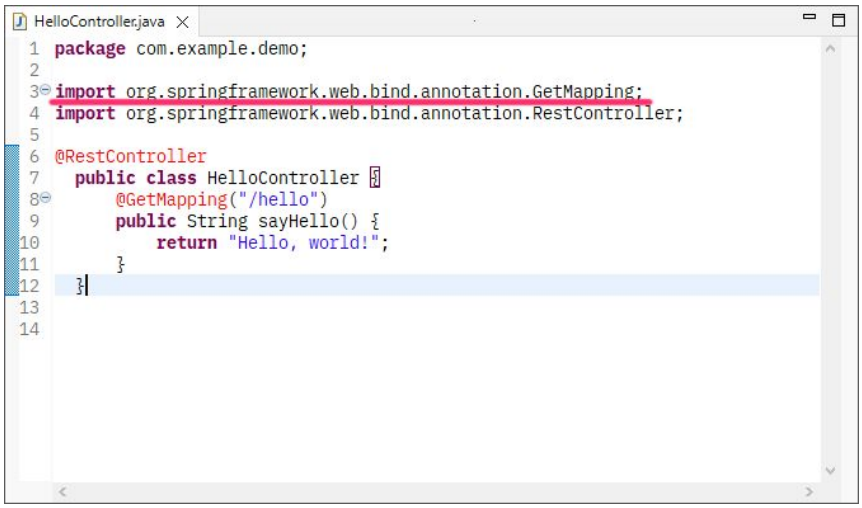


7)この中の「**'GetMapping'をインポートします**」のリンクをクリックする。



8)org.springframework.web.bind.annotation.GetMappingのimport文が挿入される。

⇒@GetMappingの赤波線も消える



クイックフィックスは、クラスやインターフェースのimport文も作成できます。手順は上記と同じ「赤波線が引かれているものにマウスカーソルを合わせる」です。

4．Thymeleafでフォーム操作

前章ではレスポンス(処理結果)として文字列をブラウザ上に表示しました。ではもっとWebアプリケーションらしくWeb画面(HTML)を返したいときはどうすればいいでしょう？技術的には文字列操作でも対応可能です。しかし、以下のようにコードの見通し・可読性が悪く、よい方法とは言えないでしょう。

【リスト4-1】文字列操作でHTMLを組み立てる

```
StringBuilder sb = new StringBuilder();
final String LS = System.lineSeparator();
sb.append("<!DOCTYPE html>" + LS);
sb.append("<html>" + LS);
sb.append("<head>" + LS);
sb.append("<meta charset=\"UTF-8\">" + LS);
  :
sb.append("</html>" + LS);
return sb.toString();
```

実際のWebアプリケーションでは、条件によって作成する内容を変えたり、繰り返し処理も必要です。またCSS、JavaScriptなどもあるでしょうから、さらに複雑になります。

このためWebアプリケーション開発では、プログラムを「入力データを処理する部分」と「Web画面を作成する部分」に分け、後者には**「テンプレートエンジン」**というものを利用するのが一般的です。

Spring Bootにも複数種類の「テンプレートエンジン」が用意されています。本書では、そのうちの１つである**「Thymeleaf(タイムリーフ)」**を使用します。

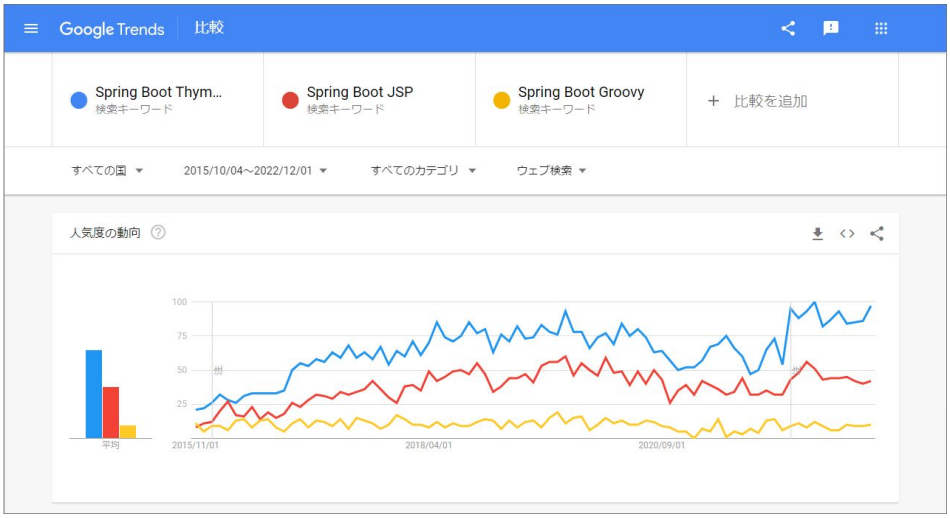
Thymeleafの特長

1．HTMLとの親和性が高い

Thymeleafで作成したWeb画面のソースコードは、一見するとHTMLファイルのようです。実際、格納するときのファイル拡張子はhtmlです。これは<input>などのHTMLタグに、Thymeleaf独自の属性を追加していくためです。HTMLの知識があれば非プログラマーでも、習得コストは比較的少なくて済む、と言われています。

2．最も使用されている

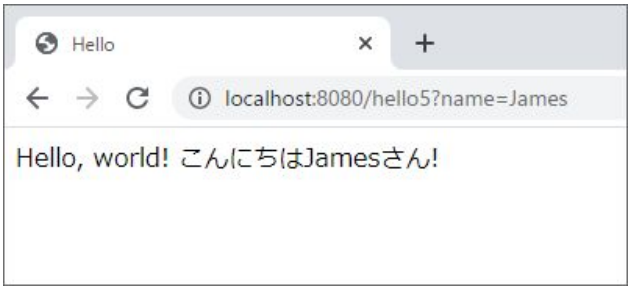
適用実績について統計的な裏付けはないのですが、Spring Bootで最も使用されているテンプレートエンジンはThymeleafではないかと思います。他にもJSP(JavaServer Pages)、Groovyなども利用可能ですが、試しにGoogle Trendsで比較してみると、注目されているのはThymeleafのようです。



本書では、HTMLがわかれば習得が容易なThymeleafを使いWeb画面を作成していきます。

4.1 ThymeleafでHello, world!

Thymeleafも"Hello, world!"から始めます。
前章Hello2のように、パラメータで渡された名前を含めて表示できるようにします。
■実行例：http://localhost:8080/hello5?name=James



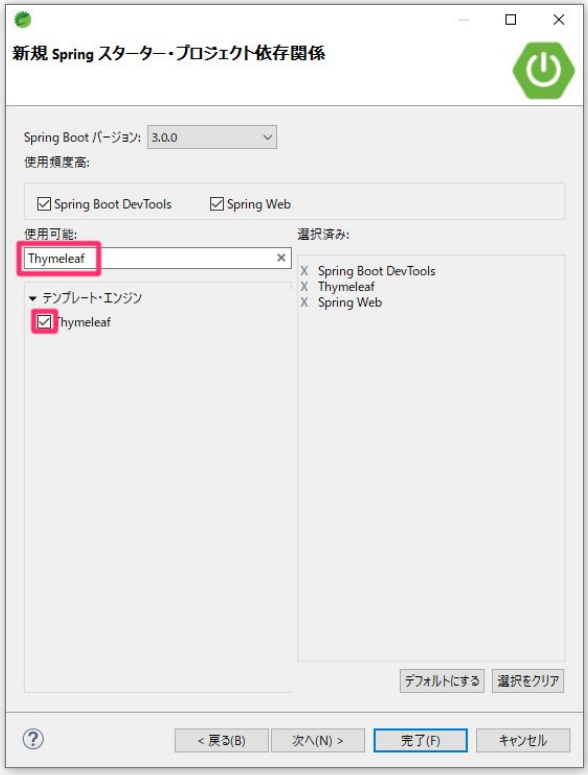
プロジェクトの作成方法は、これまでと同じです。依存関係では「Thymeleaf」を追加します。

作成するプロジェクトの仕様

プロジェクト名	Hello5
依存関係	Spring Web, Spring Boot DevTools, **Thymeleaf**
コントローラークラス	Hello5Controller
作成ファイル	src/main/resources/templates/hello.html

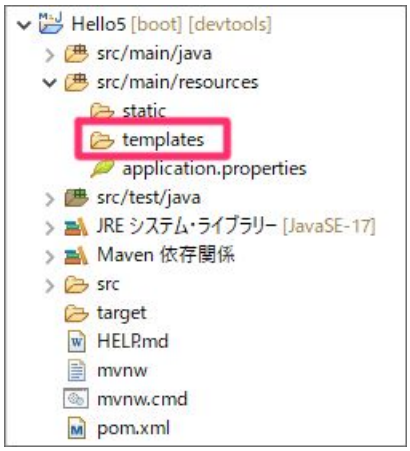
依存関係

[使用可能]に"**Thymeleaf**"と入力して選択します。



作成ファイル

Thymeleafが使用するファイル(**テンプレート**)をプロジェクトに追加します。追加場所は**src/main/resources**下にある**Thymeleafフォルダ**です。Thymeleafで使うファイルは、この**templates**下に格納します。



ここにhello.htmlというファイルを作成し、以下のように編集してください。

【リスト4-2】src/main/resources/templates/hello.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">    <!-- ① -->
<head>
<meta charset="UTF-8">
<title>Hello</title>
</head>
<body>
Hello, world! こんにちは<span th:text="${name}"></span>さん!  <!-- ② -->
</body>
</html>
```

一見するとただのHTMLですが、２つポイントがあります。

①**<html xmlns:th="http://www.thymeleaf.org">**

「このHTMLファイルに"th:～"という属性があれば、それはThymeleafのもの」ということを宣言しています。こう宣言するとThymeleafはテンプレートファイルから"th:～"を探し、そこで何らかの処理(主にコンテンツ生成)を行います。②はその典型例です。

②****

span要素に**th:text**という属性があります。Thymeleafはth:textを見つけると、右辺"${...}"の中に書かれている変数の値を、タグのテキストに変換します。この${...}を「**変数式**」と言います。

たとえば変数nameに"James"が設定されていれば、Thymeleafが次のようにします。

```
Hello, world! こんにちは<span>James</span>さん!
```

"th:～"の処理が終わったら、サーバーはThymeleafが処理したテンプレートファイルを、レスポンスとしてブラウザに返します。

では、変数nameはどこから来るのでしょうか？これはコントローラークラスから渡されます。

【リスト4-3】com.example.demo.Hello5Controller.java

```java
package com.example.demo;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

@Controller                          //  ①
public class Hello5Controller {
    @GetMapping("/hello5")
    public ModelAndView sayHello(@RequestParam("name") String name,
                                 ModelAndView mv) {  //②
        mv.setViewName("hello");     //  ③
        mv.addObject("name", name); //  ④
        return mv;                  //  ⑤
    }
}
```

前章のコントローラークラスに比べ、いくつか違っている点があります。

①@Controllerアノテーション

前章では@RestControllerをクラスに付与しましたが、@Controllerに変わっています。

@RestControllはテキスト用のものでした。一方、ここではThymeleafが処理したテンプレート(HTML)を返します。こういった場合は、@Controllerを使います。

②ModelAndViewオブジェクト

メソッド引数にModelAndViewオブジェクトが追加されています。またメソッドの戻り値もModelAndView型です。

このModelAndViewは、その名の通り「**モデル**」と「**ビュー名**」を保持するクラスです。

・ビュー名：次に表示する画面名

・モデル：ビュー(画面)で使用するデータ

ビュー名はModelAndView#setViewName()で設定します(③)。

```
mv.setViewName("hello");    //  ③
```

Thymeleafは設定されたビュー名に拡張子".html"を追加したファイルをsrc/main/resources/templates下から探します。よって③は、次に表示する画面として【リスト4-2】のhello.htmlを指定したことになります。

次のModelAndView#addObject()は、このビューが使うデータを渡します(④)。

```
mv.addObject("name", name); //  ④
```

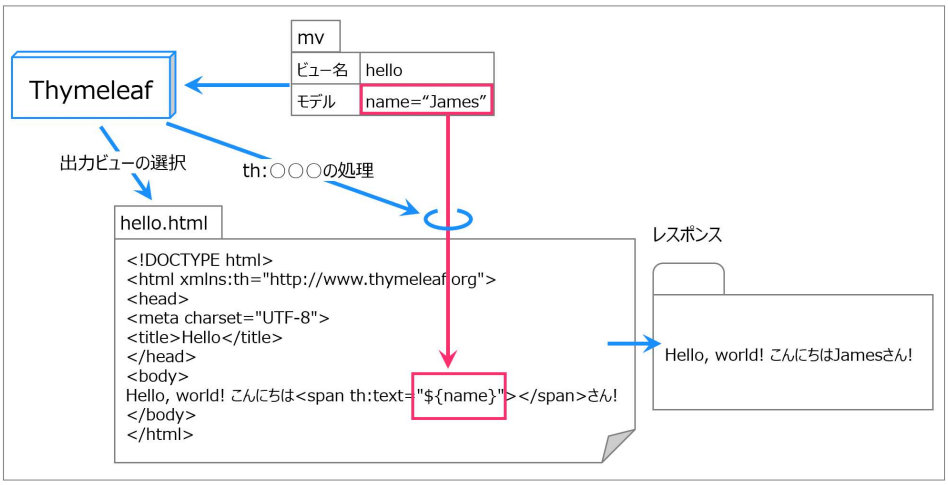
第2引数がビューに渡すデータ(オブジェクト)です。第1引数はその名前です。少々紛らわしいのですが、hello.htmlに記述したth:text="${name}"のnameに対応するのは、**第1引数の方**です。

名前を以下のように変えてもかまいません。この場合は、ビュー側も第1引数に合わせます。

```
mv.addObject("onamae", name); //  ④
    ↓
<span th:text="${onamae}"></span>
```

最後にビュー名とモデルをセットしたmvをreturnします(⑤)。mvはThymeleafに渡され、前述の処理が実行されます。そしてブラウザには、入力された名前を含むhello.htmlが返されます。

図で表すと以下のようなイメージになります。



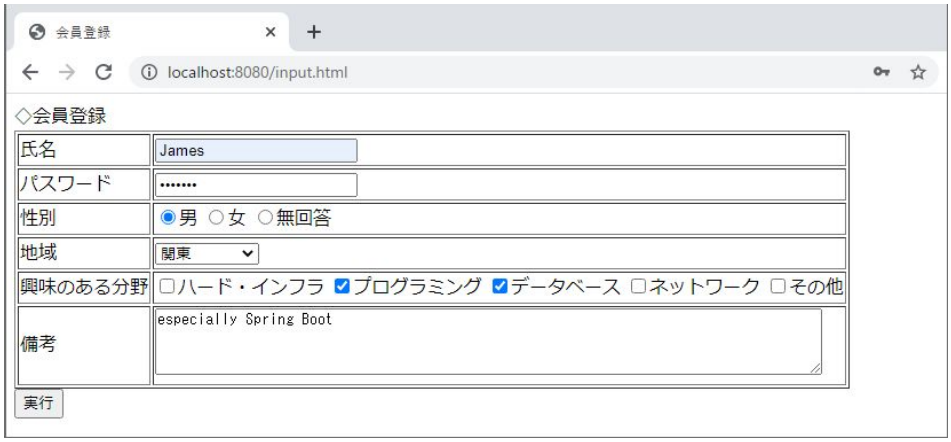
【図4-1】 Thymeleafの機能

4.2 フォーム部品

前章「3.4 POSTリクエストの処理」では、フォーム部品としてテキスト入力フィールド(type="text")とsubmitボタン(type="submit")を使いました。本節ではそれ以外のフォーム部品の使い方について説明します。

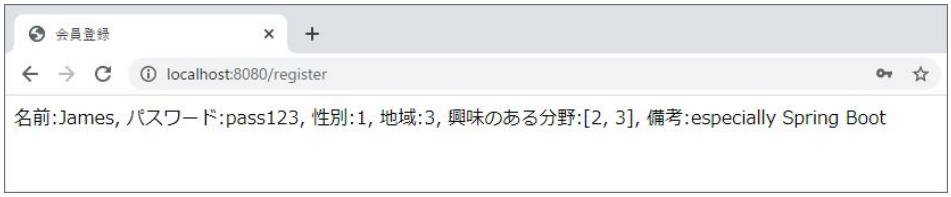
例として以下のような「会員登録画面」を作成します。データ入力後[実行]ボタンをクリックすると、入力内容が一度サーバーへ送られ、文字列として返される単純なものです。

■入力画面(input.html):http://localhost:8080/input.html



↓[実行]ボタンクリック

■結果画面(result.html)



作成するプロジェクトの仕様

プロジェクト名	Registration
依存関係	Spring Web, Spring Boot DevTools, Thymeleaf
コントローラークラス	RegistrationController
作成ファイル	src/main/resources/static/input.html(入力画面) src/main/resources/templates/result.html(結果画面)

入力画面はstaticフォルダにinput.htmlとして作成します。このファイルはth:～を含まないため、テンプレートではありません(=Thymeleafが処理しない)。こういったファイルはstaticフォルダ下に格納します。

【リスト4-4】src/main/resources/static/input.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>会員登録</title>
</head>
<body>
  ◇会員登録
  <form action="/register" method="post">
    <table border="1">
      <tr>
        <td>氏名</td>
        <td>
          <input type="text" name="name">
        </td>
      </tr>
      <tr>
        <td>パスワード</td>
        <td>
          <input type="password" name="password">
        </td>
      </tr>
      <tr>
        <td>性別</td>
        <td>
          <input type="radio" name="gender" value="1">男
          <input type="radio" name="gender" value="2">女
          <input type="radio" name="gender" value="9">無回答
```

```
      </td>
    </tr>
    <tr>
      <td>地域</td>
      <td>
        <select name="area">
          <option value="1">北海道</option>
          <option value="2">東北</option>
          <option value="3">関東</option>
          <option value="4">中部</option>
          <option value="5">近畿</option>
          <option value="6">四国</option>
          <option value="7">九州・沖縄</option>
        </select>
      </td>
    </tr>
    <tr>
      <td>興味のある分野</td>
      <td>
        <input type="checkbox" name="interest" value="1">ハード・インフラ
        <input type="checkbox" name="interest" value="2">プログラミング
        <input type="checkbox" name="interest" value="3">データベース
        <input type="checkbox" name="interest" value="4">ネットワーク
        <input type="checkbox" name="interest" value="9">その他
      </td>
    </tr>
    <tr>
      <td>備考</td>
      <td>
        <textarea name="remarks" rows="4" cols="80"></textarea>
      </td>
  </table>
  <input type="submit" value="実行">
</form>
```

```
</body>
</html>
```

下表はこのinput.htmlを含め、本書で使っているフォーム部品と属性値をまとめたものです。またform要素についても記載しました。

【表4-1】form要素

タグ	詳細
<form></form>	フォーム部品の上位要素として記述する。 action属性：入力内容の送信先 method属性：入力内容の送信方法(post, getのいずれか)

【表4-2】フォーム部品

タグ	詳細
<input type="text">	テキスト入力フィールド(1行) value属性：入力フィールドのテキスト(パラメータ値) placeholder属性：初期表示する文字列
<input type="password">	パスワード入力フィールド value属性：入力フィールドのテキスト(パラメータ値)
<input type="radio">	ラジオボタン(１つのみ選択可能) 同じname属性値のラジオボタンでグループを構成する value属性：選択されているとき、サーバーへ送信する値(パラメータ値) checked属性：画面表示時、選択状態とする
<input type="checkbox">	チェックボックス(複数選択可能) 同じname属性値のチェックボックスでグループを構成する value属性：選択されているとき、サーバーへ送信する値(パラメータ値) checked属性：画面表示時、選択状態とする
<input type="submit">	フォームに入力された内容をform要素のaction属性/method属性に 従い送信する。

<input type="hidden">	画面上には表示されないが値を保持する項目(隠し項目) value属性：非表示で保持するテキスト(パラメータ値)
<textarea></textarea>	テキスト入力フィールド(複数行) rows属性：入力フィールドの幅(文字数で指定) cols属性：入力フィールドの高さ(行数で指定) placeholder属性：初期表示する文字列
<select></select>	セレクトボックス 選択肢はoption要素で作成する size属性： 選択肢の表示行数 省略した場合 multiple属性がある場合、4 ない場合、1(プルダウン形式) multiple属性： 選択肢を複数選択可能にする ⇒Windowsの場合、[Shift]または[Ctrl]キーを押しながら選択 ない場合、選択できる選択肢は１つ
<option></<option>	選択肢 value属性：選択されているとき、サーバーへ送信する値(パラメータ値) selected属性：画面表示時、選択状態とする
<button></button>	汎用ボタン type属性："button"を指定すると上記input要素のtype="submit"と同じく送信ボタンになる formaction属性：form要素のaction属性をこの属性値で上書き formmethod属性：form要素のmethod属性をこの属性値で上書き

※name属性はoption要素以外で指定可能。サーバー送信時、パラメータ名として使われる。

> これらは「代表的なもの」です。他にもいろいろなフォーム部品や属性があります。興味がある方は調べてみてください。

次は会員登録フォームに入力された値を受け取るコントローラークラスです。
入力/選択されたデータは、POSTリクエストでも「パラメータ名=パラメータ値」形式、つまり「name属性の値=入力/選択されたデータ」のペアで送られます。よってハンドラーメソッドは@RequestParamを使います。

【リスト4-5】com.example.demo.RegistrationController.java

```
package com.example.demo;

import java.util.Arrays;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class RegistrationController {
    @PostMapping("/register")
    public ModelAndView register(@RequestParam("name") String name,
                                 @RequestParam("password") String
password,
                                 @RequestParam("gender") int gender,
                                 @RequestParam("area") int area,
                                 @RequestParam("interest") int[]
interests,
                                 @RequestParam("remarks") String remarks,
                                 ModelAndView mv) {
        StringBuilder sb = new StringBuilder();
        sb.append("名前：" + name);
        sb.append("，パスワード：" + password);
        sb.append("，性別：" + gender);
        sb.append("，地域：" + area);
        sb.append("，興味のある分野：" + Arrays.toString(interests));
```

```
        sb.append(", 備考：" + remarks.replaceAll("\n", ""));

        mv.setViewName("result");                    //  ①
        mv.addObject("registData", sb.toString()); //  ②
        return mv;
    }
}
```

@PostMapping("/register")は、input.htmlの<form action="/register" method="post">に対応しているので、[実行]ボタンがクリックされるとregister()が呼び出されます。

なおinput.htmlの「性別」「地域」に対応するメソッド引数gender,areaはint型です。これらの入力値はint型に変換された後、バインドされます。
また「興味のある分野」のinterestsはint型の配列(int[])です。チェックボックスは選択肢の数だけチェック可能なので、値の個数は可変です。そこでバインド先は配列とします。

あとは前節のHello5Controllerと同様です。ビュー名を"result"とし(①)、"registData"という名前で処理結果を渡します(②)。
ビュー側は、単純にこの内容をspan要素として表示します。

【リスト4-6】src/main/resources/templates/result.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>会員登録</title>
</head>
<body>
  <span th:text="${registData}"></span>
</body>
</html>
```

4.3 @ModelAttributeを使った入力データの取得

前節のRegistrationでは、フォーム部品に対応する引数をハンドラーメソッドに並べていました。しかしフォーム部品が多くなると面倒です。またフォーム部品の値は、クラスのプロパティとして表すのが一般的です。このようなクラスは、フォームの内容を保持するので**「フォームクラス」**などと呼ばれています。

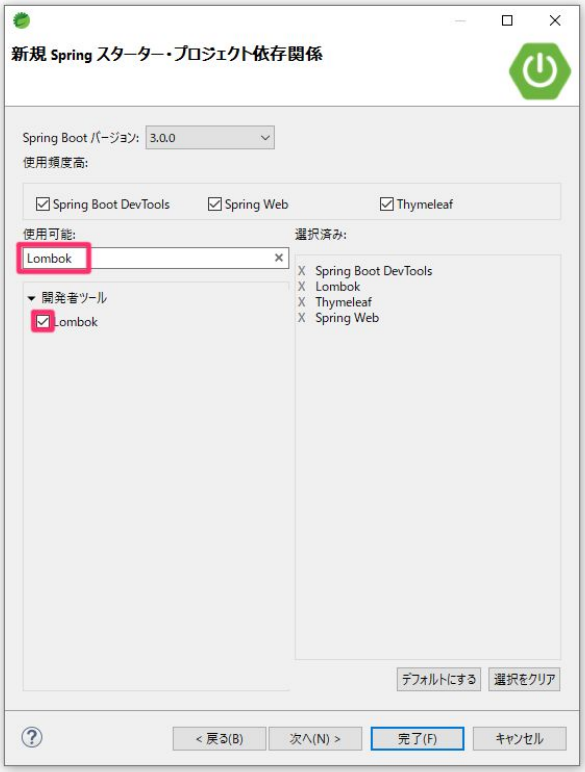
Spring Bootには、こういったデータを扱う**@ModelAttribute**というアノテーションがあります。これにLombokを組み合わせるとフォームからのパラメータ取得が簡潔になります。

作成するプロジェクトの仕様

プロジェクト名	Registration2
依存関係	Spring Web, Spring Boot DevTools, Thymeleaf, **Lombok**
作成クラス	Registration2Controller

依存関係

依存関係に「Lombok」を追加します。[使用可能]に**"Lombok"**と入力して選択してください。



なお

・入力画面(src/main/resources/static/input.html)

・結果画面(src/main/resources/templates/result.html)

は、前節Registrationと同じものを使用しますので、このRegistration2へコピーしてください。

⇒操作方法は本章最後を参照してください。

まずフォームの内容を受け取るフォームクラスを作成します。

【リスト4-7】com.example.demo.RegistData.java

```
package com.example.demo;
import lombok.Data;

@Data                    //  ①
public class RegistData {
    private String name;
    private String password;
    private int gender;
```

```
    private int area;
    private int[] interest;
    private String remarks;
}
```

フォームクラスには、フォーム部品と**同じ名前のプロパティ**を定義します。データ型は入力される値を考慮して決めます。

そしてこれらプロパティに対するsetter/getterメソッドは、Lombokの**@Dataアノテーション**で作成します(①)。このアノテーションを付与すると、デフォルトコンストラクタやtoString()などのメソッドも、合わせて自動生成します。

STSのアウトラインを見ると、自動生成されたメソッドを確認することができます。

> STSにもgetter/setter生成機能がありますが(メニュー [ソース(S)] > [getterおよびsetterの生成(R)...])、これはソースコードを生成します。それに対しLombokはソースコードを変えず、直接classファイルを操作します。

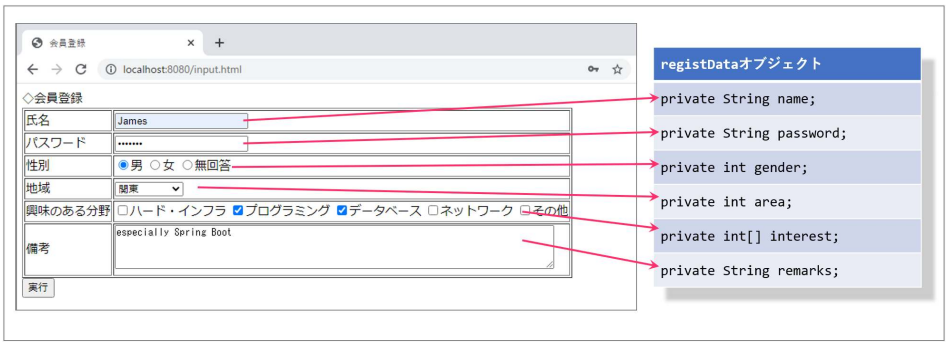
このフォームオブジェクトを、ハンドラーメソッドの引数に追加します(①)。

【リスト4-8】com.example.demo.Registration2Controller.java

```
package com.example.demo;
```

```
import java.util.Arrays;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.ModelAndView;

@RestController
public class Registration2Controller {
    @PostMapping("/register")
    public ModelAndView register(@ModelAttribute RegistData registData,  //
①
                                 ModelAndView mv) {
        StringBuilder sb = new StringBuilder();
        sb.append("名前：" + registData.getName());
        sb.append(", パスワード：" + registData.getPassword());
        sb.append(", 性別：" + registData.getGender());
        sb.append(", 地域：" + registData.getArea());
        sb.append(", 興味のある分野：" +
Arrays.toString(registData.getInterest()));
        sb.append(", 備考：" + registData.getRemarks().replaceAll("\n",
""));

        mv.setViewName("result");
        mv.addObject("registData", sb.toString());
        return mv;
    }
}
```

@ModelAttributeを付与すると、フォーム部品のname属性値とフォームクラスのプロパティ名をキーにして、フォーム部品の値がフォームオブジェクトにバインドされます。こうするとフォーム部品ごとの引数を並べる必要が無いのでシンプルになります。



【図4-2】フォームオブジェクトへのバインド

バインドの結果は、同じ名前のフォーム部品とフォームクラスのプロパティがあるか、ないかで以下のように変わります。

【表4-3】@ModelAttributeによるバインド

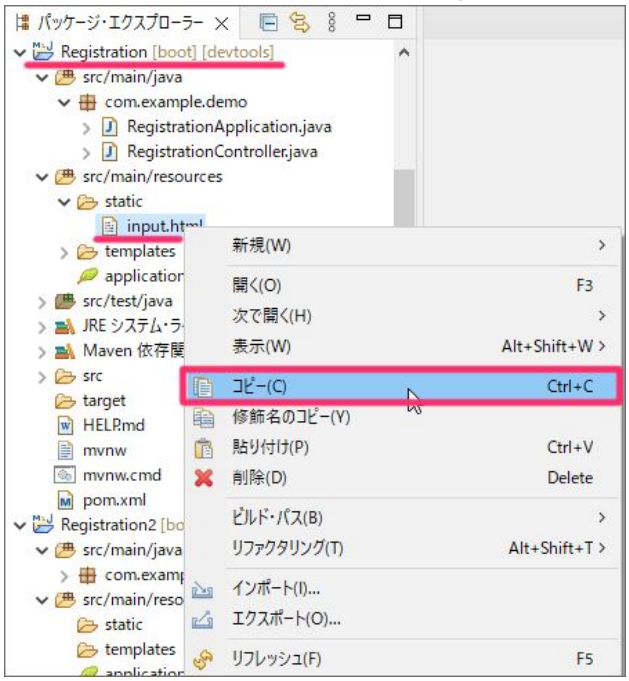
フォーム部品	フォームクラスプロパティ	結果
あり	あり	フォーム部品の値がフォームオブジェクトにバインドされる。
あり	なし	フォームオブジェクトには何もバインドされない
なし	あり	実行時エラーになる

本章ではThymeleafの簡単な紹介と、各種フォーム部品の使い方を説明しました。本書の後半では、これらを駆使してWeb画面を作成していきます。

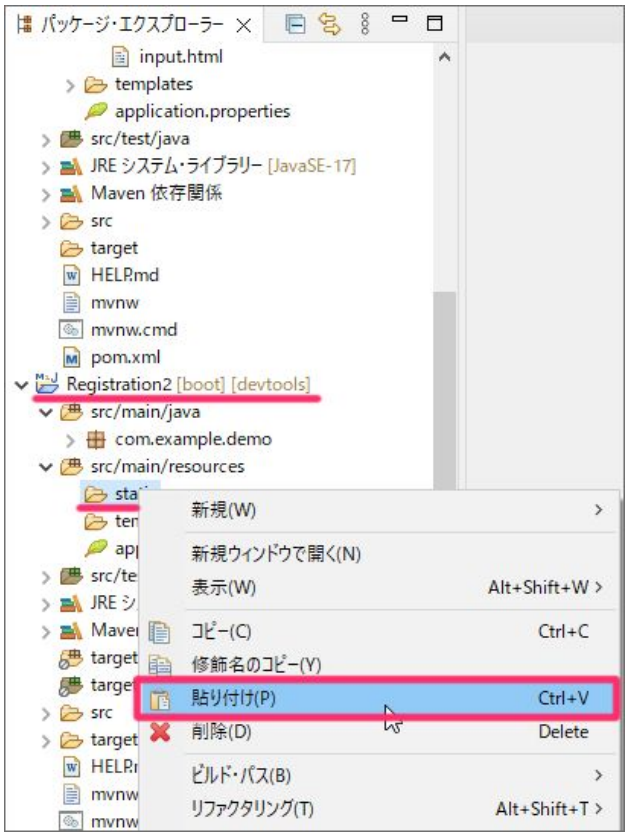
補足：プロジェクト間でファイルをコピーする

ここではプロジェクトRegistrationのsrc/main/resources/static/input.htmlをRegistration2へコピーする操作を例に説明します。

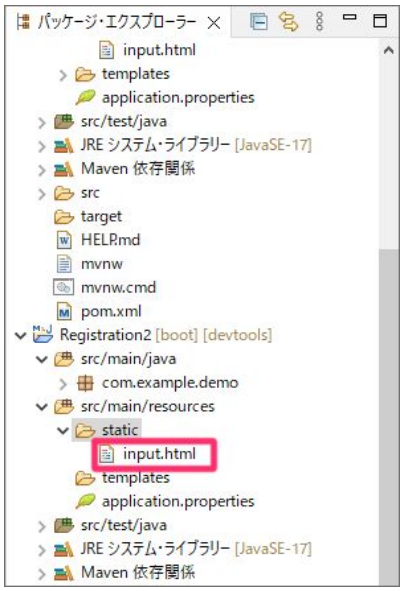
1)パッケージ・エクスプローラーでコピー元ファイルRegistrationの src/main/resources/static/input.htmlを右クリック > [コピー(C)]を選択する。



2)コピー先のRegistration2のsrc/main/resources/staticを右クリック > [貼り付け(P)]を選択する。



3)input.htmlがコピーされる。



■参考

・パッケージ・エクスプローラー内では、ドラッグ＆ドロップ操作でファイルを移動させることもできます。

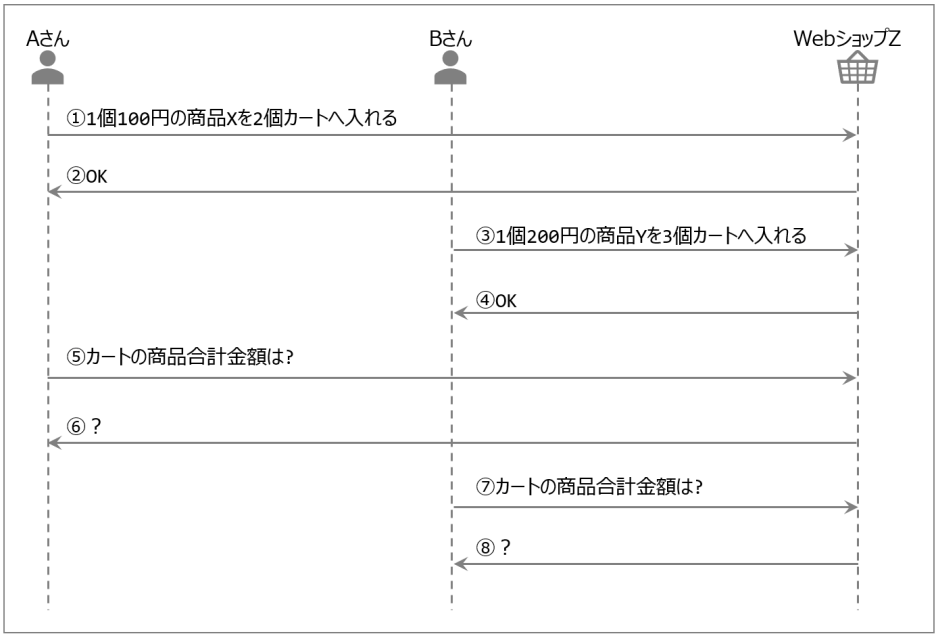
・パッケージ・エクスプローラーには、Windowsのエクスプローラーでコピーしたファイルを貼り付けることもできます。

5．セッション操作

5.1 セッションの必要性

本書ではWebアプリケーションのクライアントソフトとして、ブラウザを使用しています。そしてブラウザは、HTTPというプロトコルを使い、サーバーとリクエスト/レスポンスをやり取りします。これはインターネット上のショッピングサイトでも同じです。

例えば「インターネット上にZというWebショップがあり、Aさん、Bさんが気に入った商品をカートに入れた」とします。シーケンス図で表すと次のような状況です。



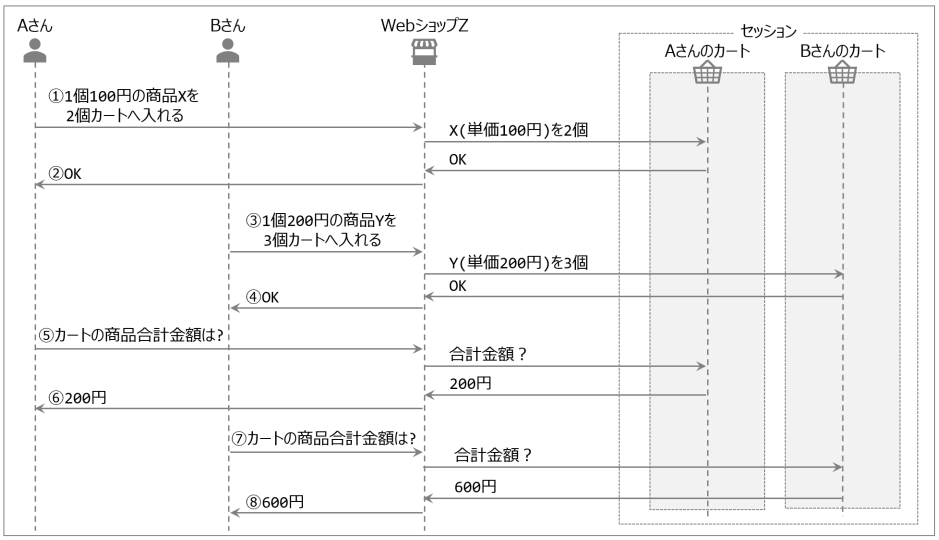
【図5-1】カートがユーザーごとに用意されていない？

⑤でAさんが、カートの合計金額を照会します。⑥はいくらでしょう？当然200円と思うでしょう。しかしZのシステムが、適切に作成されていないと800円が返ってくるかもしれません。これはAさんとBさんの商品が同じカートに入っている、ということです。

実はHTTPプロトコルだけでは、①はAさんのリクエスト、③はBさんのリクエスト、ということがわかりません。そのためZのサーバーは同じカートに入れてしまったわけです。もちろんこれでは困ります。①と③は異なる人(ブラウザ)からのリクエストと認識し、カートは別なものとしなければなりません。

こういったとき必要になるのが**セッション(Session)**です。セッションを使えば「誰からのリクエストか？」を識別でき、さらにその人ごとの専用領域を持つことができます。カートはこの領域の中に作ればいいのです。

図で表すと次のようなイメージです。



【図5-2】ユーザーごとのカートをセッションに作成する

本書には、複数ユーザーで同時に実行するプログラム例は載せていません。だからといってセッションが不要、というわけではありません。画面をまたいでデータを共有するときは、そのデータをセッション内に保存する必要があります。これはHTTPプロトコルが、**前のリクエスト/レスポンスの結果を保持しない**からです。このため「最初に入力した内容を、次の次の画面で使う」といった場合は、データをセッションへ格納し、後から取得できるようにします。

5.2 数あてゲームでセッションを学ぶ

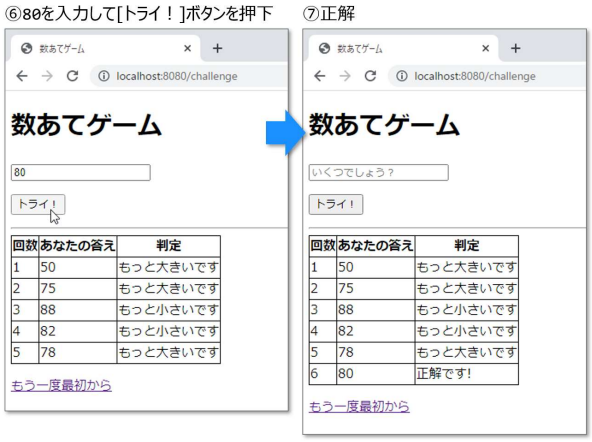
本章ではセッションを使い「画面間でデータを共有する方法」について説明します。
例として、簡単な「数あてゲーム」を作成します。

数当てゲームのルール

(1)ゲーム開始時、サーバーは1～100の中から数字を1つ選び、それを「正解」とする。
(2)ユーザーは(1)の正解を推測し、その数字を「回答」として入力する。
(3)コンピューターは「正解」と「回答」を比較し、以下のようなメッセージを表示する。
(3.1)正解 < 回答の場合 ：「もっと小さいです」と表示する。
(3.2)正解 = 回答の場合 ：「正解です！」と表示する。
(3.3)正解 > 回答の場合 ：「もっと大きいです」と表示する

実行例

上記のルールをもとにSpring Bootを使い、Webアプリケーションにした例を以下に示します。ブラウザからhttp://localhost:8080/へアクセスすると、ゲーム開始です。
本章では、このプログラムを作成して行きます。



【図5-3】数あてゲームの実行例

上記の操作をシーケンス図で表すと以下のようになります。

【図5-4】数あてゲーム実行例のシーケンス

最初のリクエストで正解を作成します。HTTPプロトコルだけでは、この正解を次のリクエストまで保持できません。そこでセッションに格納し、回答が送られてくる都度、セッションの正解と比較します。

作成するプロジェクトの仕様

プロジェクト名	Game

依存関係	Spring Web, Spring Boot DevTools, Thymeleaf, Lombok
コントローラークラス	GameController
作成ファイル	src/main/resources/templates/game.html

数あてゲームのコントローラークラスは、以下のようになっています。

【リスト5-1】com.example.demo.GameController.java

```
package com.example.demo;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import jakarta.servlet.http.HttpSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class GameController {

  @Autowired            //  ①
  HttpSession session;   //  ①

  @GetMapping("/")
  public String index() {
    session.invalidate();                    //  ②
    // 答えを作ってSessionに格納
    Random rnd = new Random();                //  ③
    int answer = rnd.nextInt(100) + 1;        //  ③
    session.setAttribute("answer", answer);  //  ④
```

```
    System.out.println("answer=" + answer);    //コンソールに正解を出力する(^^)
    return "game";
  }

  @PostMapping("/challenge")                    //  ⑤
  public ModelAndView challenge(@RequestParam("number") int number,
                                ModelAndView mv) {  //⑥
    // セッションから答えを取得
    int answer = (Integer)session.getAttribute("answer");  //⑦

    // ユーザーの回答履歴を取得
    @SuppressWarnings("unchecked")
    List<History> histories = (List<History>)session.getAttribute("histories");//⑧
    if (histories == null) {
      histories = new ArrayList<>();
      session.setAttribute("histories", histories);
    }
    // 判定→回答履歴追加
    if (answer < number) {
      histories.add(new History(histories.size() + 1, number, "もっと小さいです"));//⑨

    } else if (answer == number) {
      histories.add(new History(histories.size() + 1, number, "正解です！"));    //⑨

    } else {
      histories.add(new History(histories.size() + 1, number, "もっと大きいです"));//⑨
    }

    mv.setViewName("game");                 //⑩
    mv.addObject("histories", histories);  //⑩
```

```
    return mv;
  }
}
```

このプログラムのポイントは次の2行です(①)。

```
  @Autowired            //  ①
  HttpSession session;   //  ①
```

HttpSession(jakarta.servlet.http.HttpSession)がセッションのクラスです。変数sessionには、正解や回答履歴を格納して行きます。しかし、sessionを初期化する処理が見当たりません。このままでは実行時エラー(例外NullPointerException)になるのでは？そう思う方がいるかもしれません。

その答えが**@Autowiredアノテーション**にあります。通常HttpSessionオブジェクト(=セッション)は、リクエストを表すHttpServletRequest(jakarta.servlet.http.HttpServletRequest)オブジェクトからgetSession()で取得します。@Autowiredを付与すると、コントローラークラス起動時、これに相当する処理が自動的に行われ、sessionに設定されます。つまり**自分でセッションのインスタンスをセットする必要がない**、というわけです。

> この初期化方法は「**フィールドインジェクション**」と呼ばれています。以前はよく使われていましたが、現在は「**コンストラクタインジェクション**」という方法が推奨されています。後者については次章で説明します。

> Spring Bootでセッションを操作する場合、@SessionAttributeアノテーションや@Scopeアノテーションを使う方法もあります。興味がある方は調べてみてください。

ハンドラーメソッドは２つ定義しています。このようにコントローラークラスには、複数のハンドラーメソッドを定義できます。

【表5-1】GameControllerクラスのハンドラーメソッド

アノテーション	ハンドラーメソッド	発生タイミング
@GetMapping("/")	index()	1)初期表示

		2)リンク[もう一度最初から]をクリックしたとき
@PostMapping("/challenge")	challenge()	[トライ！]ボタンをクリックしたとき

index()の処理内容

index()はURLパス"/"がGETリクエストで送られてきたときのハンドラーメソッドです。発生するタイミングは、上表のように2パターンあります。

1)の初期表示は、ブラウザからhttp://localhost:8080/がアクセスされた場合です(シーケンス図参照)。

2)のリンク[もう一度最初から]は、後述するように<a href="/">としています。これもhttp://localhost:8080/へのアクセスとなります。

index()の処理は、最初にinvalidate()を呼び出し、セッション内の情報をクリアします(②)。これは2)の[もう一度最初から]をクリックしたとき用の処理です。1)の初期表示では、セッションにまだ何も格納していないので、実行しても状態は変わりません。

```
    session.invalidate();                    //  ②
```

正解は、次の2行で作成します(③)。

```
  Random rnd = new Random();                  //  ③
  int answer = rnd.nextInt(100) + 1;          //  ③
```

java.util.Random#nextInt()は0～引数を超えない範囲でint型の乱数を返します。nextInt(100)なら、0～99になるので、+1して正解の範囲を1～100とします。

これをHttpSession#setAttribute()でセッションに格納します(④)。

```
  session.setAttribute("answer", answer);    //  ④
```

第2引数が格納する値(オブジェクト)、第1引数がその名前です。setAttribute()の第2引数はjava.lang.Object型と定義されているため、answerはInteger型へAutoboxingさ

れてから、セッションに格納されます。

最後に"game"をreturnして次画面を指定します。

```
return "game";
```

最初にアクセスされたとき、コントローラーから画面に渡すデータはありません。つまりaddObject()は不要です。Thymeleafには、次に表示するビュー名を伝えるだけです。

こういった場合、上記のように文字列でビュー名を指示できます。これでModelAndView#setViewName()と同じように、templates下にある".html"を付加した名称のファイルが、次画面としてブラウザへ送信されます。このgame.htmlは、後述【リスト5-3】にあります。

challenge()の処理内容

challenge()は[トライ！]ボタンをクリックしたときの処理です。アノテーションから、

・POSTリクエストでURLパスは**/challenge**(⑤)

・回答は**number**という名前のパラメータ(⑥)

ということが推測できると思います。実際、後述する画面game.html(【リスト5-3】)はそのようになっています。

```
  @PostMapping("/challenge")    //  ⑤
  public ModelAndView challenge(@RequestParam("number") int number,
                                ModelAndView mv) {  //⑥
```

このメソッドでは、まずセッションに保存されている正解をHttpSession#getAttribute()で取得します(⑦)。

```
  int answer = (Integer)session.getAttribute("answer");  //⑦
```

引数の名前は、前述のsetAttribute()で格納したときのもの(="answer")です。

getAttribute()の戻り値はjava.lang.Object型なので、格納時Autoboxingされた Integer型へキャストします。これがanswerに代入されるときAuto-Unboxingされてint型となります。

次に前回までの回答履歴をセッションから取得します(⑧)。

```
  List<History> histories =
(List<History>)session.getAttribute("histories");//⑧
  if (histories == null) {
      histories = new ArrayList<>();
      session.setAttribute("histories", histories);
  }
```

回答履歴はHistory型オブジェクトを要素としたArrayListとしています。ゲーム(再)開始直後は、セッションに回答履歴は無いためnullが返されます。この場合、回答履歴を生成し、"histories"という名前でセッションに格納します。

この histories = new ArrayList<>() のhistories はList<History>型です。ArrayListはListインターフェースを実装しているので代入できます。さらにhistoriesに対しては、Listで宣言されているメソッドしか使わないことを表しています。

右辺のArrayList<>は、この場合ArrayList<History>の略です。ダイアモンド演算子(<>)が型推論により、左辺の<History>からHistory型と推論し、これを割り当てます。

そして正解(answer)と回答(number)を比較し、結果を回答履歴に追加します(⑨)。

```
  histories.add(new History(histories.size() + 1, number, "もっと小さいです"));//⑨
  histories.add(new History(histories.size() + 1, number, "正解です！"));
//⑨
  histories.add(new History(histories.size() + 1, number, "もっと大きいです"));//⑨
```

最後にビュー名と処理結果(回答履歴)をModelAndViewオブジェクトに設定して(⑩)、returnします。

```
  mv.setViewName("game");                  //⑩
  mv.addObject("histories", histories);  //⑩
  return mv;
```

> この説明ではindex()/challenge()でやっていることがわからない、という方のためにパラパラ漫画風の解説を章末に載せています。上記処理を理解する上で欠かせない「参照」の考え方からセッション操作まで、ステップ・バイ・ステップで説明していますので、そちらも参考にしてください。

個々の回答を表すHistory型クラスは、以下のように定義しています。

【リスト5-2】com.example.demo.History.java

```
package com.example.demo;

import lombok.AllArgsConstructor;
import lombok.Getter;

@AllArgsConstructor
@Getter
public class History {
  private int seq;
  private int yourAnswer;
  private String result;
}
```

ここでもLombokのアノテーションを使っています。自動生成するコードは下表の通りです。

【表5-2】Lombokのアノテーション

アノテーション	生成対象
@AllArgsConstructor	全フィールドへ値をセットするコンストラクタ
@Getter	フィールドに対するgetterメソッド

このうち@AllArgsConstructorは、Historyの場合、以下のようなコンストラクタを作成します(ソースコード上には現れません)。これは回答履歴を追加するとき使用しています(⑨)。

```
  public History(int seq, int yourAnswer, String result) {
    super();
    this.seq = seq;
    this.yourAnswer = yourAnswer;
    this.result = result;
  }
```

結果を表示する画面は、以下のようになっています。

【リスト5-3】src/main/resources/templates/game.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>数あてゲーム</title>
<style>
th, td {
      border: solid 1px; /* 枠線指定 */
}

table {
      border-collapse: collapse; /* セルの線を重ねる */
}
</style>
</head>
<body>
  <h1>数あてゲーム</h1>
  <form action="/challenge" method="post">
    <p>
      <input type="text" name="number" placeholder="いくつでしょう？">
    </p>
    <p>
      <input type="submit" value="トライ！">
    </p>
  </form>
  <hr>
  <table>
    <tr>
      <th>回数</th>
      <th>あなたの答え</th>
      <th>判定</th>
    </tr>
```

```
    <tr th:each="h:${histories}">              <!-- ① -->
      <td th:text="${h.seq}"></td>             <!-- ① -->
      <td th:text="${h.yourAnswer}"></td>      <!-- ① -->
      <td th:text="${h.result}"></td>          <!-- ① -->
    </tr>
  </table>
  <p>
    <a href="/">もう一度最初から</a>
  </p>
</body>
</html>
```

ポイントは回答履歴を作成する**th:each属性**を追加したtr要素、およびその子要素tdです(①)。

th:eachはJavaの拡張for文と同じように、コレクションオブジェクトの要素を処理するために使います。

右辺は「変数:${コレクションオブジェクト}」という意味です。

<tr>～</tr>は、このコレクションオブジェクトの要素数分、作成されます。

変数は、コレクションオブジェクトの各要素を表します。変数に対して処理を記述すると、それがコレクションオブジェクトの全要素に適用されます。

ここではコレクションオブジェクトがhistoriesとなっています。これはGameControllerが、画面に渡したオブジェクトの名前です(⑩)。

```
  mv.addObject("histories", histories); //⑩
```

historiesはList<History>型なので、変数hはHistory型です。このため${h.seq}は、History型オブジェクトのseqプロパティと解釈され、内部でh.getSeq()が呼び出されます。後続の${h.yourAnswer},${h.result}も同様です。これらの結果をtd要素で表示します。

> th:textの代わりに[[...]]というインライン式を使うこともできます。以下のように書いても同じ結果が得られます。
>
> ```
>   <tr th:each="h:${histories}">
> ```

```
    <td>[[${h.seq}]]</td>
    <td>[[${h.yourAnswer}]]</td>
    <td>[[${h.result}]]</td>
  </tr>
```

以上で数あてゲームをプレイできるようになります。
プレイしながら「最小手数で正解にたどり着くにはどうすればよいか？」を考えてみるのも面白いでしょう。

補足：セッション操作詳説

Java言語の導入教育をしていると、「参照」で悩む方が数多く見受けられます。C言語の経験者には「ポインタ演算ができないポインタ」という説明をすると、ある程度納得してくれます。しかしセッションと組み合わせると、やはり難しいようです。

そこで「参照」というものを説明した上で、それがGameControllerのセッション操作でどのように使われているか、パラパラ漫画風に、ソースコードを1行ずつ解説して行きます。

参照とは？

Java言語のデータ型には**「基本データ型(プリミティブ型)」**と**「参照型」**があります。

基本データ型は、以下の8種類です。プログラムで追加できません。

byte, short, int, long, float, double, char, boolean

参照型は、以下に該当するものです。Stringクラスなどはこの典型です。無数にあり、プログラムで追加できます。

クラス、インターフェース、配列

ここでは基本データ型のintと、参照型のjava.lang.Integerクラスを使って説明して行きます。どちらも32ビットの整数値を表すことができます。まず100という整数値を格納する変数を作成します。

```
int int100 = 100;
Integer integer100 = new Integer(100);
```

「変数は箱のようなもの」という説明をよくしますが、それでいくとint100は、以下のようなイメージです。

いまint100という箱を開けると中から100が取り出せます。このように「箱の中に**値そのもの**が入っている」のが基本データ型の変数です。

一方integer100は少々事情が異なります。

箱の中に100はありません。あるのは100のありかを表す特別な値(ここでは●とします)です。箱を開けたら「それは下駄箱の上から2段目、左から3番目の中だよ」と書かれた紙が入っていた、という感じです。このように参照型の変数には「**値の場所**を示す特別な値が格納されている」と思ってください。

では100はどこへ行ったのか？専門的になりますが**「ヒープエリア」**という領域に置かれています。つまり●は、ヒープエリアの100の場所を表しているわけです。

> C言語なら●はアドレスです。しかしJavaでは、アドレスがプログラマから隠ぺいされています。そしてポインタ演算に相当する機能はありません。

> int100やinteger100のようなローカル変数は「スタック領域」(あるいは「ローカル領域」)と呼ばれるところに作成されます。つまりint型など基本型の値は、スタック領域上に格納されています。

`Integer integer100 = new Integer(100)` をもっと分解してみます。
この1行は以下のように書いても同じです。

```
Integer integer100;
integer100 = new Integer(100);
```

最初の行は、変数integer100の宣言です。宣言直後はnullです。

2行目の右辺はIntegerオブジェクトを生成します。newはヒープエリアに作成すること、と考えてもよいでしょう。

これを代入演算子(=)でinteger100に代入するわけですが、ここで代入されるのは生成したIntegerオブジェクトの場所を表す特別な値●です。

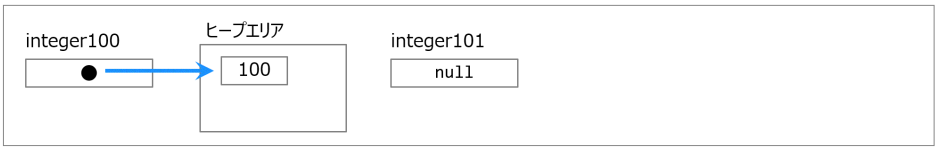
結果このようになります。

integer100は100の場所を「指し示している」わけです。「どこにあるの？」と聞かれたら「あそこだよ」と答えられるようになっています。これが**「参照」**です。以後、参照を青矢印線で表します。

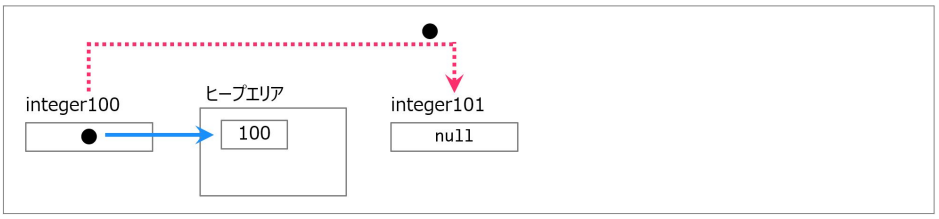
ここからさらに次の2行を実行します。これも分解しながら説明します。

```
Integer integer101 = integer100;
integer100 = new Integer(200);
```

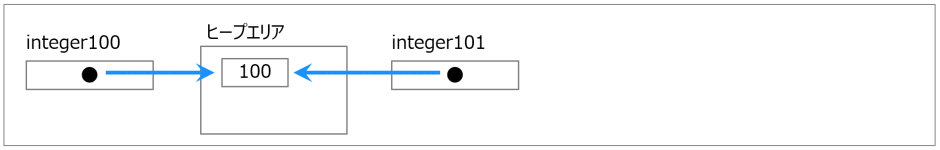
まず1行目の左辺でInteger型の変数integer101を作成します。



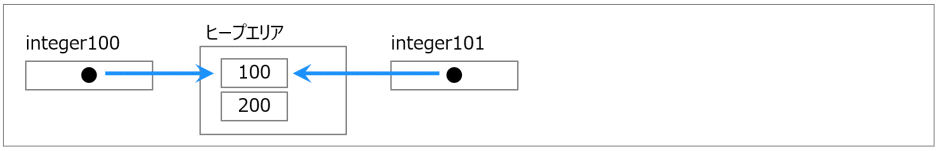
ここにinteger100を代入します。integer100の値は、100のありかを示す特別な値●ですが、これがinteger101に代入されます。



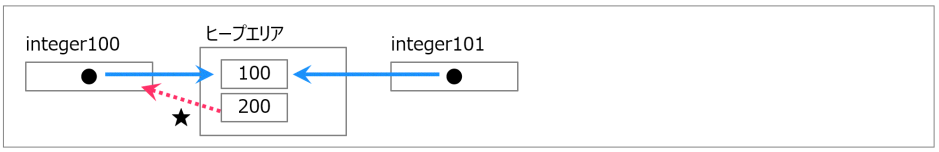
結果、integer101も100を指し示すことができるようになります。この「**特別な値が代入されると、それも参照になる**(=指し示すことができるようになる)」というのが、参照を理解する、もう1つのポイントです。



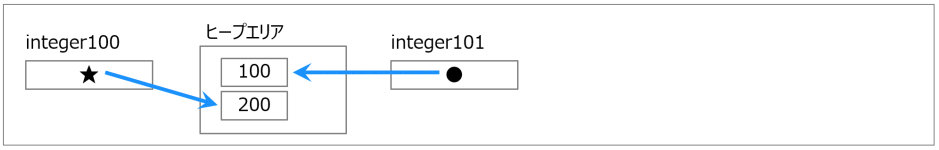
次に2行目の右辺で200というInteger型オブジェクトを生成します。



これをinteger100に代入するわけですが、●ではありません。●は100の場所です。200の場所は、●とは異なる特別な値になります。ここでは★とします。



これがinteger100に代入されることで、200を参照します。



どうでしょうか？「参照」という言葉に慣れてきたでしょうか？

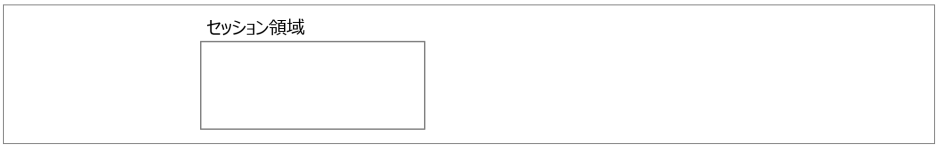
基本型変数は値を直接表すが、参照型変数は間接的に表す、という考え方です。ここまでの説明を参考に、参照の働きを自分なりにイメージできるようにしてください。

次節からはGameControllerの動作を1行ずつ解説します。以降も「参照」を青矢印、「参照の代入」を赤点線で表します。

数当てゲーム 起動時の処理

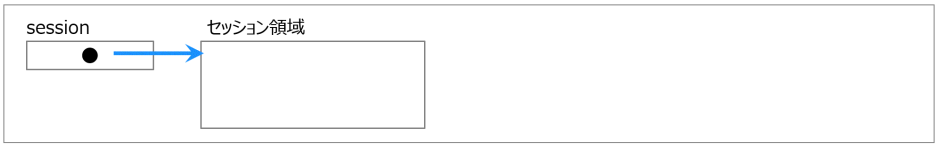
①http://localhost:8080/アクセス⇒サーバー到着

この時点で、アクセスしてきたブラウザ用にセッション領域が作成されます。最初は何も格納されていません。



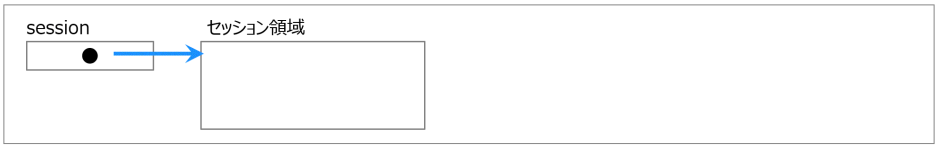
②GameController起動時

@Autowiredにより変数sessionがセッション領域を参照します。本編で解説しましたが、セッションというのはHttpSessionクラス型のオブジェクトです。セッションもクラスなので参照になるわけです。



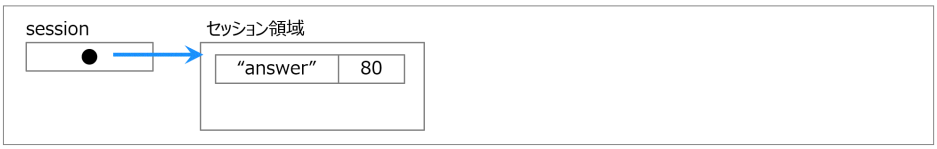
③index()開始

session.invalidate()を実行しますが、この時点では何も格納されていないため、変化はありません。



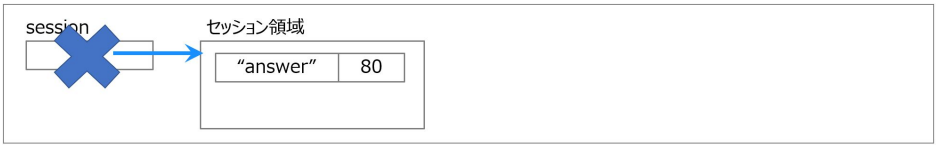
④session.setAttribute("answer", answer)

セッション領域に“answer”という名称で、正解(本編の実行例では80)を格納します。これはsessionが「参照しているところ」、別な言い方をすると「指し示しているところ」にsetAttribute()を実行する、という意味になります。

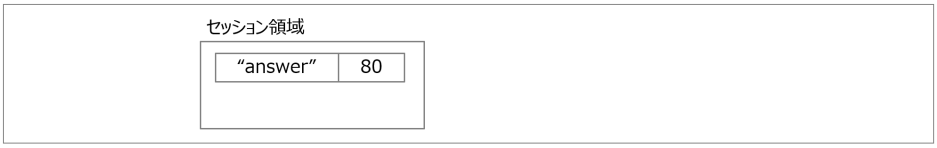


⑤index()終了⇒GameController終了

変数sessionはGameControllerクラスのインスタンス変数なので、GameController終了時点で消去されます。しかし**セッション領域は残ります**。



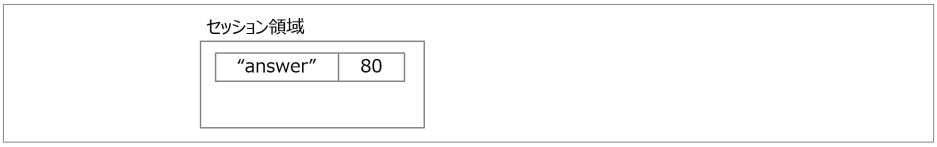
よって以下のような状態になります。



1回目の処理

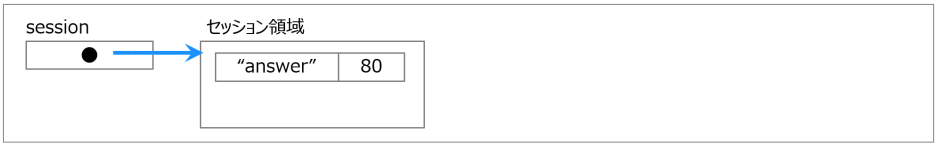
1回目の回答として50を入力して[トライ！]ボタンをクリックしたとします。

①セッション領域は正解だけが格納された状態で残っています。



②GameController起動

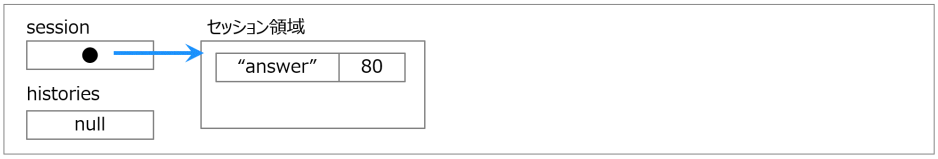
@Autowiredにより変数sessionが保存されていたセッション領域を参照します。これで**前項④の状態を復元**できました。つまり、前回の終了時点から、処理を継続できるようになったわけです。



③challenge()開始

④List<History> histories =
(List<History>)session.getAttribute("histories");

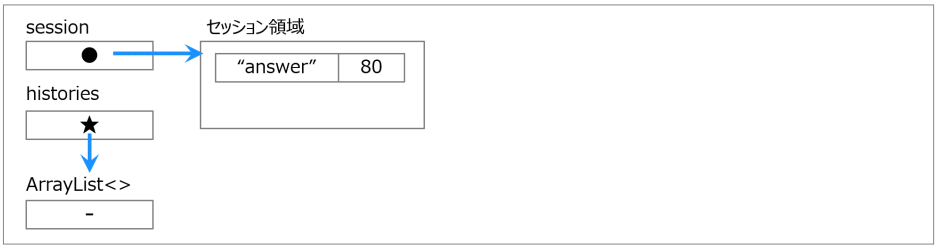
セッション領域から”histories"という名称のデータ（回答履歴）を取得します。しかしこの時点では存在しないので、historiesには null が設定されます。



⑤histories = new ArrayList<>();

回答履歴が無いのでArrayList<History>オブジェクトとして作成し、histories から参照できるようにします。

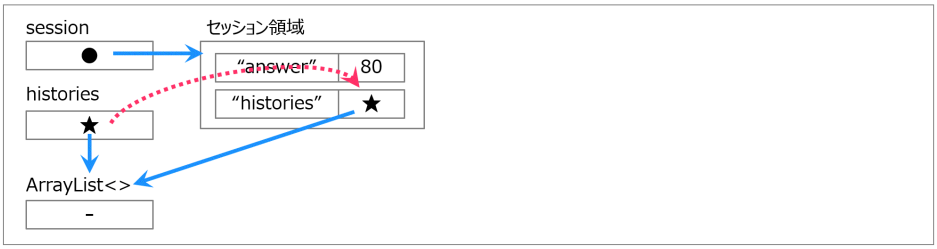
※以下、図中ではArrayList<History>をArrayList<>と表記します。



⑥session.setAttribute("histories", histories);

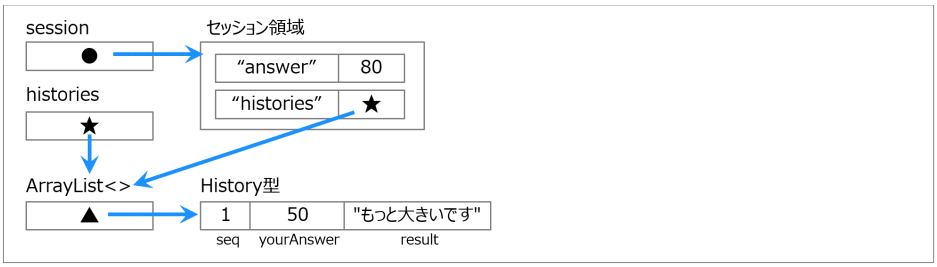
⑤で作成したhistoriesを"histories"という名称でセッションに保存します。値はArrayList<History>オブジェクトの場所を表す特別な値★です(赤点線)。

ArrayList<History>オブジェクトは、2か所から参照されています。



⑦histories.add(new History(histories.size() + 1, number, "もっと小さいです"));

1回目の判定結果をhistoriesに追加します。ここでは最初にHistoryオブジェクトをnew(生成)します。その結果、また新しい特別な値▲ができます。historiesにadd()しているのは、この▲です。

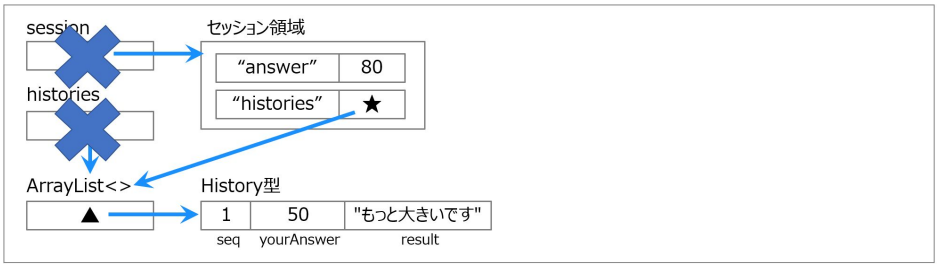


⑧challenge()終了⇒GameController終了

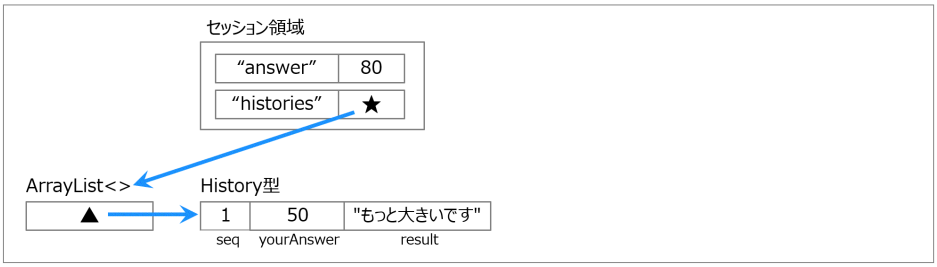
historiesはchallenge()内で宣言されたローカル変数なので、challenge()終了時に消去されます。

sessionもGameController終了時に消去されます。

しかしセッション領域は残ります。そして**セッション領域から参照されているオブジェクトも残ります。**



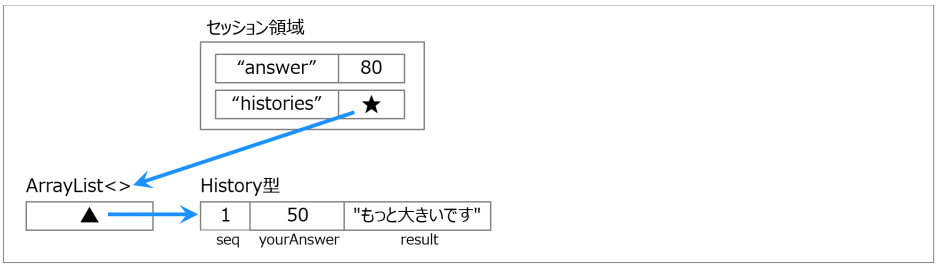
よって以下のような状態になります。



2回目の処理

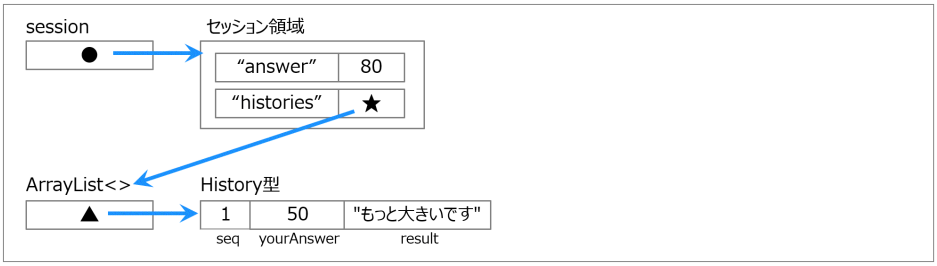
2回目の回答として75を入力して[トライ！]ボタンをクリックしたとします。

①セッション領域は1回目終了時点の状態で残っています。



②GameController起動

@Autowiredにより変数sessionがセッション領域を参照します。

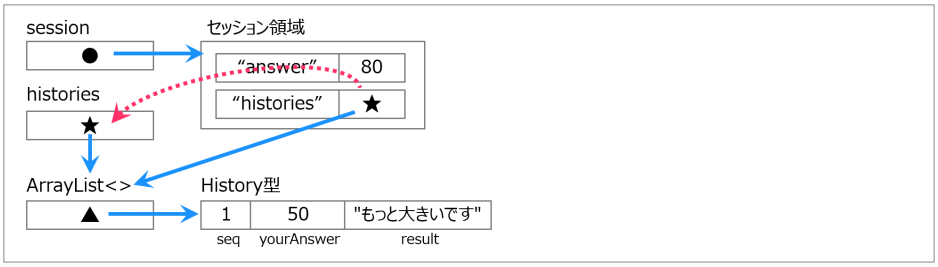


③challenge()開始

④List<History> histories =
(List<History>)session.getAttribute("histories");

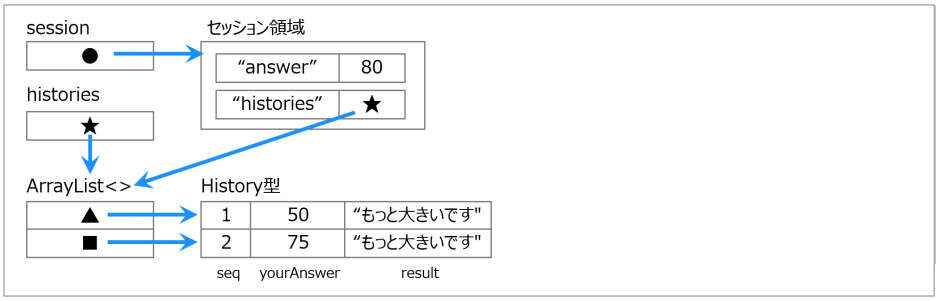
セッション領域から”histories"という名称のデータ（回答履歴）を取得します。これで値★がhistoriesに設定されます。★は1回目に格納したArrayList<History>オブジェクトの場所であり、これでhistoriesからも参照できるようになります。これで**前項⑦と同じ状態に**できました。

このように毎回、前の終了時点から、処理を継続できるようにしています。



⑤histories.add(new History(histories.size() + 1, number, "もっと小さいです"));

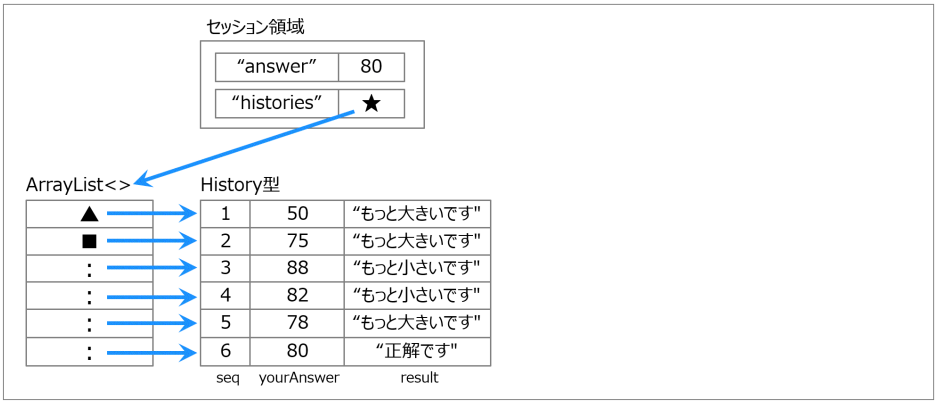
2回目の判定結果をhistoriesに追加します。



⑥以降は、1回目と同じです。

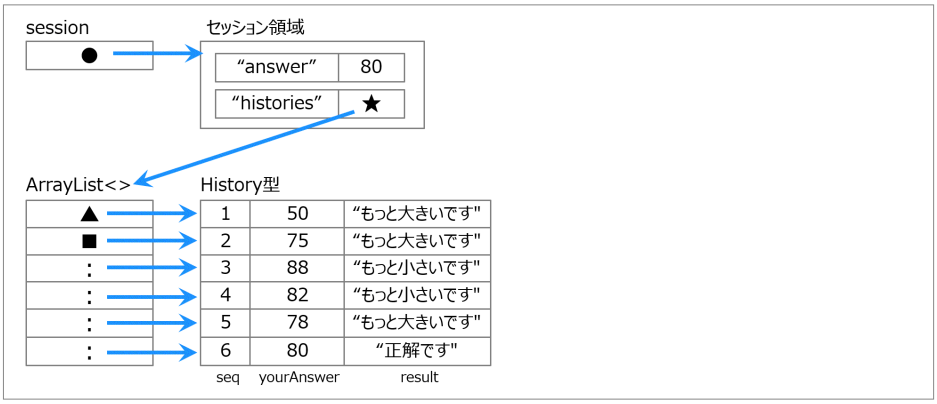
[もう一度最初から]クリック時

①セッション領域には前回の正解と回答履歴が残っています。例えば以下のような状態です。



②GameController起動

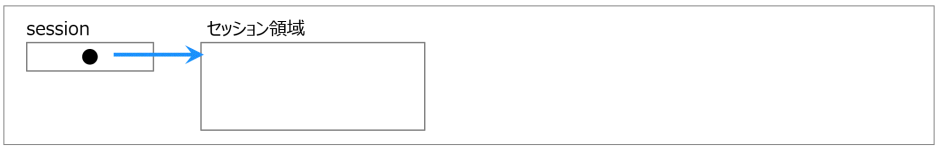
@Autowiredにより変数sessionがセッション領域を参照します。



③index()開始

④session.invalidate()

　sessionが参照しているセッション領域の内容をクリアします。



　これで「起動時の処理②」と同じ状態になります。

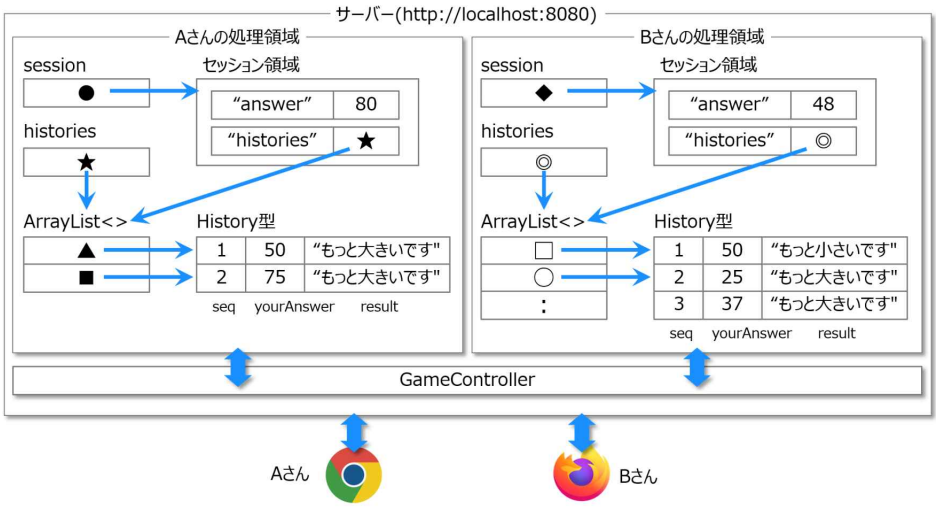
> 正確に言うと、セッション領域から参照されていたArrayList<History>は、まだメモリ上(ヒープ領域)に存在しています。しかしどこからも参照されていないので、アクセス不可の状態です。このように参照されていない領域は、再利用できるようにする「ガベージコレクション」という処理の対象になります。

> セッションも一定時間経過すると、削除されてしまいます。これを「セッション・タイムアウト」と呼んだりします。興味がある方は調べてみてください。

同時に実行した場合

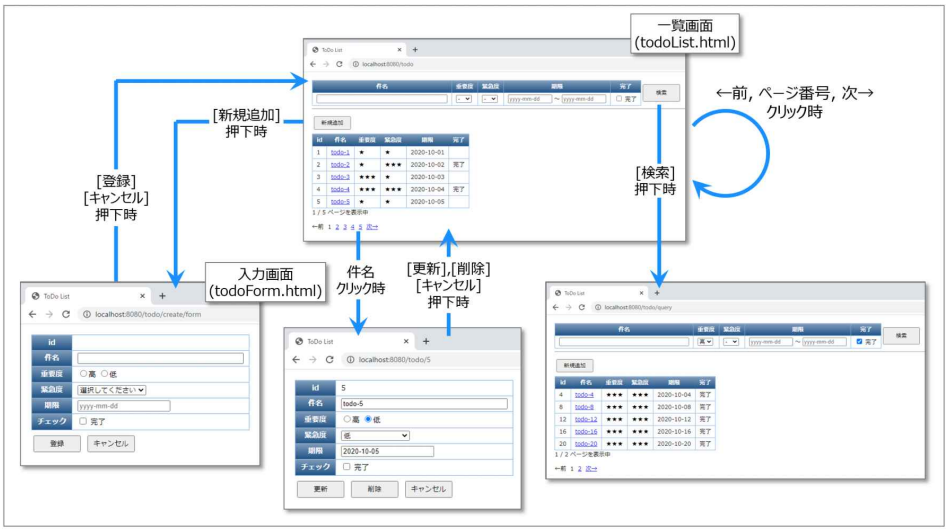
　この数当てゲームを複数人で同時に実行したらどうなるでしょうか？もちろん、きちんと動作します。試してみたい方はChromeとFireFox(あるいはEdge)を起動して、http://localhost:8080/へアクセスしてみてください。それぞれのブラウザで、数当てゲームをプレイできます。

　これはセッションにより、ブラウザごとに異なる領域が使われているためです。図で表すと以下のようなイメージです。これを見ると、本章冒頭のカートを人(ブラウザ)によって別々にする、という意味がわかるのではないかと思います。


サーバー(http://localhost:8080)
Aさんの処理領域
session
セッション領域
"answer"
80
"histories"
histories
ArrayList<>
History型
1
50
"もっと大きいです"
2
75
"もっと大きいです"
seq
yourAnswer
result
Bさんの処理領域
session
セッション領域
"answer"
48
"histories"
histories
ArrayList<>
History型
1
50
"もっと小さいです"
2
25
"もっと大きいです"
3
37
"もっと大きいです"
seq
yourAnswer
result
GameController
Aさん
Bさん

6．テーブルのデータを一覧表示する

本章からはSpring Bootを使い１つのWebアプリケーションを作成していきます。テーマは「ToDo管理」です。非常にシンプルですが、データベース上のToDoを検索、追加、更新、削除できるようになっています。



【図6-1】本書で作成するToDoアプリケーションの概要

このToDo管理アプリケーションを、章ごとに「ステップ・バイ・ステップ」で作ります。各章で追加する機能は、以下の通りです。

6章　ToDoの一覧表示

7章　ToDoの登録

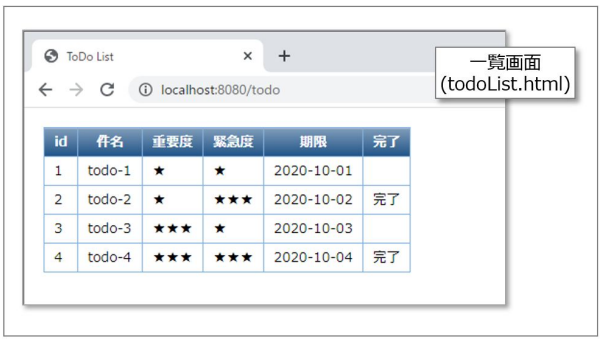
8章　ToDoの変更、削除

9,10章　ToDoの検索

11章　検索結果のページネーション(ページング)

本章では最初の一歩として、データベースに格納されているToDoをブラウザに一覧を表示します。

■本章で作成する一覧画面の実行例　⇒　http://localhost:8080/todoへアクセスする


ToDo List
localhost:8080/todo
一覧画面
(todoList.html)
id
件名
重要度
緊急度
期限
完了
1
todo-1
★
★
2020-10-01
2
todo-2
★
★★★
2020-10-02
完了
3
todo-3
★★★
★
2020-10-03
4
todo-4
★★★
★★★
2020-10-04
完了

6.1 テーブルの作成

ToDo管理で使用するテーブルは、下表の「todoテーブル」１つだけです。
「重要度」「緊急度」は別テーブルにすべき、という考えもありますが、シンプルさを優先して、この１つにしました。

【表6-1】todoテーブルの形式

列名	内容	データ型	制約	備考
id	1～	SERIAL	PRIMARY KEY	連番(自動採番)
title	件名	TEXT		
importance	重要度	INTEGER		0:低，1:高
urgency	緊急度	INTEGER		0:低，1:高
deadline	期限	DATE		
done	完了	TEXT		'Y':完了，'N':未完了

データ型のTEXT，INTEGER，DATEは、それぞれ文字列型、整数型、日付型(時分秒は持たない)の意味です。
id列のSERIALも整数型ですが、これはレコードを追加するとき、自動的に値が設定されます。デフォルトでは1，2，3...というように、前の値に+1したものがセットされます。いわゆる「連番(自動採番)」のデータ型です。
制約のPRIMARY KEY(主キー)は、「この列が同じ値のレコードは格納できない」ことを表しています。たとえばid=10のレコードがあるのに、さらにid=10のレコードを登録しようとすると、PostgreSQLがエラーにします(追加できない)。
これでid=10のレコードは、todoテーブル中に高々1件(1件、または0件)しか存在しないことを保証します。

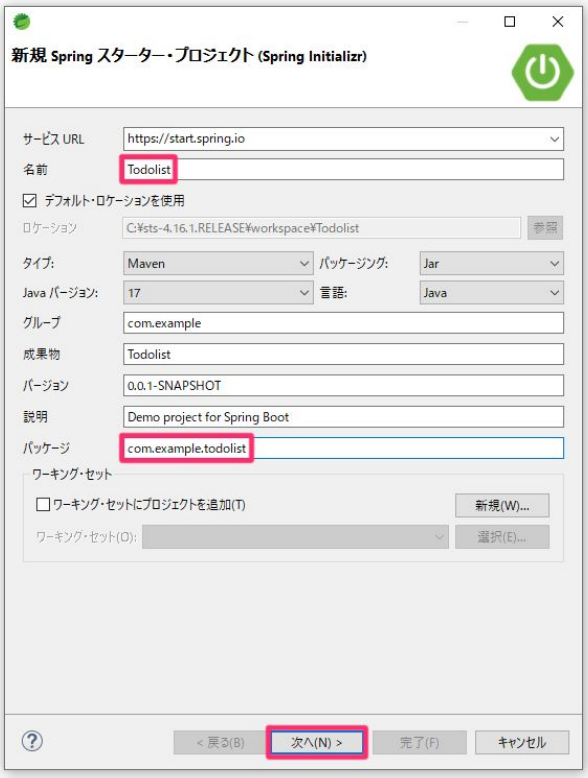
todoテーブルの作成手順は本章の最後に載せていますので、そちらを参照してください。テーブルができたら一覧表示するプロジェクトを作成していきます。

6.2 プロジェクトの構成

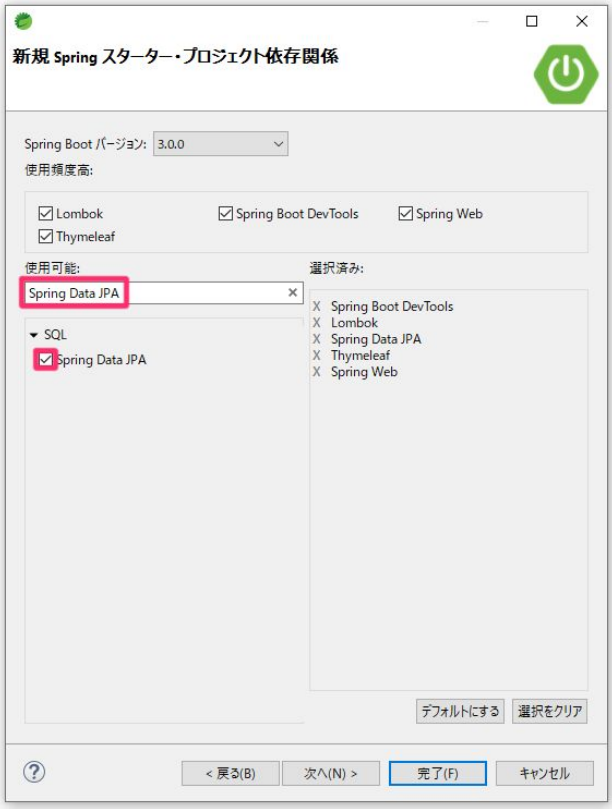
作成するプロジェクトの仕様

プロジェクト名	Todolist
パッケージ	**com.example.todolist**
依存関係	Spring Web, Spring Boot DevTools, Thymeleaf, Lombok, **Spring Data JPA, PostgreSQL Driver**
コントローラークラス	TodolistController
作成ファイル	src/main/resources/static/css/style.css src/main/resources/templates/todoList.html(ToDo一覧画面)

本章からクラスやインターフェースを管理しやすくするため、適宜パッケージ名(**com.example.todolist**)付与していきます。

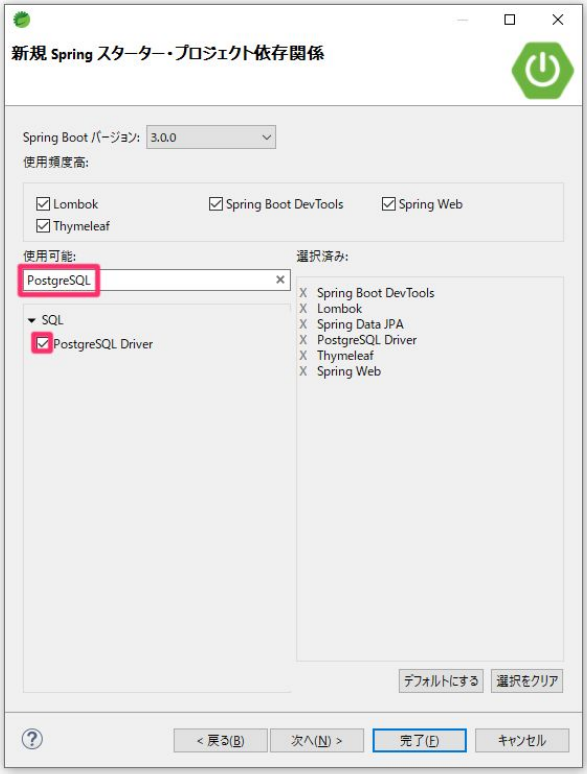


また依存関係にSpring Data JPA, PostgreSQL Driverも追加します。



JPAはJava Persistence APIの略です。Persistenceは日本語で「永続化」、簡単に言えば「データベースにデータを格納する」という意味です。Spring Data JPAを使うとJavaオブジェクトへの操作(メソッド)が、自動的にSQL文へ変換されます。つまりSQL文を書くことなく、テーブルをアクセスできるようになります。

PostgreSQL Driverは文字通り、JavaプログラムからPostgreSQL(データベース)へ接続するためのドライバーソフトです。



プロジェクトを作成したら、以下のパッケージを追加します。

【表6-2】作成パッケージ

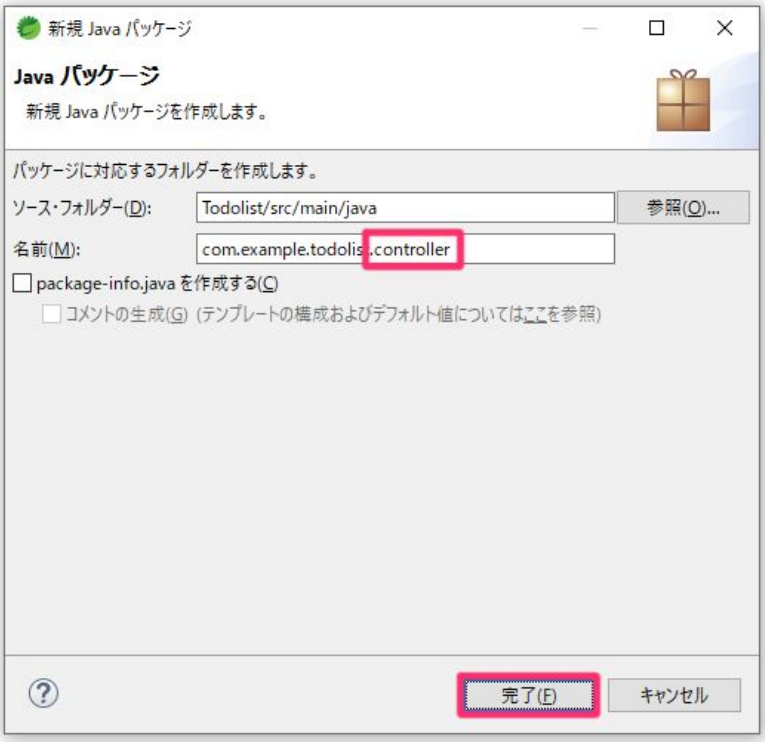
パッケージ名	用途
com.example.todolist.controller	コントローラー(controller)クラス用
com.example.todolist.entity	エンティティ(entity)クラス用
com.example.todolist.repository	リポジトリ(repository)インターフェース用

コントローラーはすでに説明しました。エンティティ、リポジトリについては、この章で説明します。

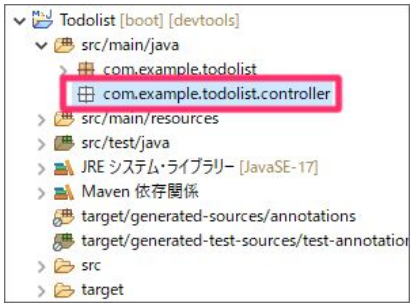
パッケージ作成手順

1)パッケージ・エクスプローラーのcom.example.todolistを右クリック > [新規(W)] > [パッケージ]を選択する。

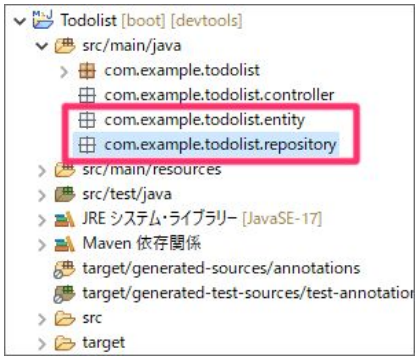
2)「新規Javaパッケージ」ダイアログの[名前(M)]を"**com.example.todolist.controller**"にする(表示されているパッケージ名に".controller"を追加する) > [完了(F)]ボタンをクリックする。



3)パッケージcom.example.todolist.controllerが追加される。



4)同様の手順でcom.example.todolist.entity, com.example.todolist.repository を作成する。



作成したパッケージに、以下のクラス/インターフェースを追加します。

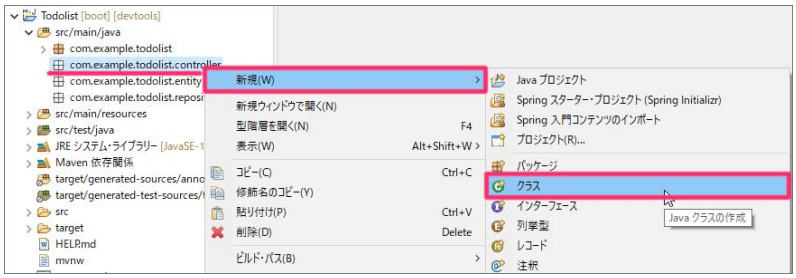
【表6-3】作成クラス/インターフェース

パッケージ名	追加するクラス/インターフェース
com.example.todolist.controller	クラス TodoListController(コントローラー)
com.example.todolist.entity	クラス Todo(エンティティ)
com.example.todolist.repository	インターフェース TodoRepository(リポジトリ)

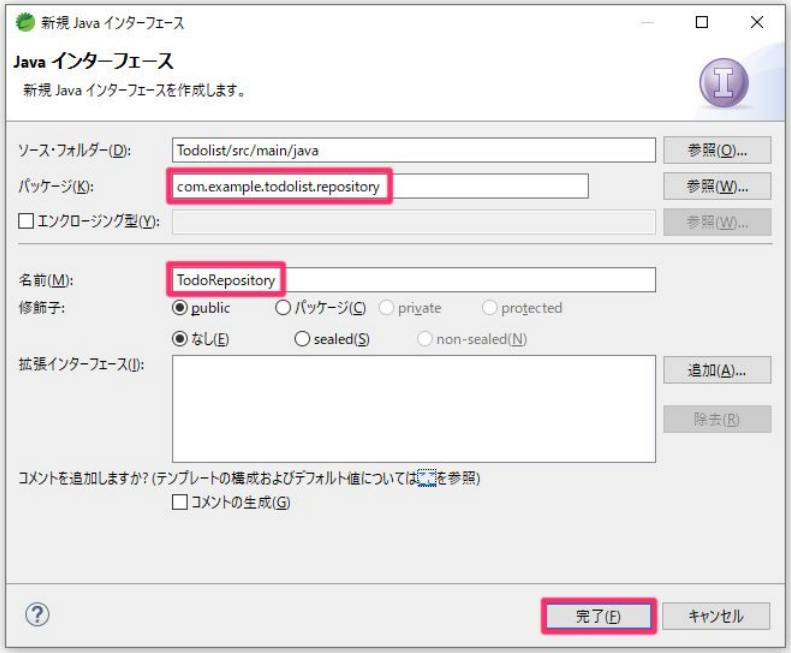
クラス/インターフェース作成手順

1)クラス

クラスを追加するパッケージを右クリック > [新規(W)] > [クラス]を選択すると、「新規Javaクラス」ダイアログの[パッケージ(K)]に、右クリックしたパッケージ名がセットされます。

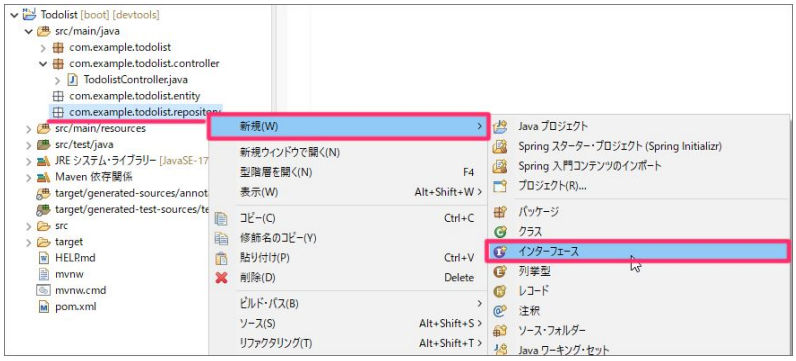


この状態でクラス名を[名前(M)]に入力 > [完了(F)]をクリックします。

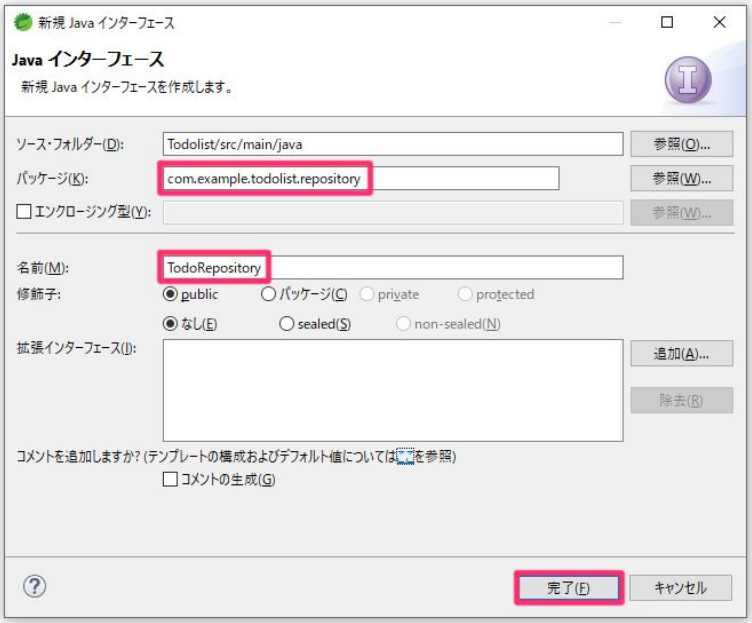


2)インターフェース

インターフェースも同様です。インターフェースを追加するパッケージを右クリック > [新規(W)] > [インターフェース]を選択すると、「新規Javaインターフェース」の[パッケージ(K)]に、右クリックしたパッケージ名がセットされます。



この状態でインターフェース名を[名前(M)]に入力 > [完了(F)]をクリックします。



ここまでの操作でsrc/main/javaは、以下のような構造になります。

```
Todolist
└ src/main/java
   ├ com.example.todolist
   │  └ TodolistApplication.java
   ├ com.example.todolist.controller
   │  └ TodoListController.java
   ├ com.example.todolist.entity
   │  └ Todo.java
   └ com.example.todolist.repository
      └ TodoRepository.java
```

6.3 エンティティ

最初にTodo.javaを以下のように編集します。

【リスト6-1】com.example.todolist.entity.Todo.java

```
package com.example.todolist.entity;

import java.sql.Date;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;

@Entity                              //  ①
@Table(name = "todo")                 //  ②
@Data
public class Todo {
    @Id                              //  ③
    @GeneratedValue(strategy = GenerationType.IDENTITY) //  ④
    @Column(name = "id")              //  ⑤
    private Integer id;

    @Column(name = "title")
    private String title;

    @Column(name = "importance")
    private Integer importance;

    @Column(name = "urgency")
    private Integer urgency;
```

```
    @Column(name = "deadline")
    private Date deadline;

    @Column(name = "done")
    private String done;
}
```

Todoクラスのプロパティは、todoテーブル(【表6-1】)の列と１対１で対応しています。つまりTodoオブジェクト１つでtodoテーブルの1レコードを表せます。このようにテーブルのレコードを表すクラスを、Spring Bootでは**「エンティティ(Entity)クラス」**と呼びます。

【表6-4】Todoクラスとtodoテーブルの関係

Todoクラス			todoテーブル		
プロパティ名	データ型	アノテーション	列名	データ型	制約
id	Integer	@Id, @GeneratedValue	id	SERIAL	PRIMARY KEY
title	String		title	TEXT	
importance	Integer		importance	INTEGER	
urgency	Integer		urgency	INTEGER	
deadline	java.sql.Date		deadline	DATE	
done	String		done	TEXT	

※すべてのプロパティに付与している@Columnは省略しています。

プロパティのデータ型は、テーブル列のものと対応させます。

todoテーブルのdeadline列は、年月日だけのDATE型なので、Todoクラスのdeadlineプロパティはjava.util.Dateではなく、java.sql.Dateとします。

Todoクラスで使っているアノテーションは次の通りです。

①**@Entity**

・このクラスがエンティティであることを示す。

②**@Table**

・このエンティティに対応付けるテーブルを指定する。

⇒これによりTodoオブジェクトへの操作は、自動的にtodoテーブルのレコードに対する操作となる(後述)。

③**@Id**

・テーブルの主キーに対応するプロパティであることを表す。

⇒プロパティ名はid以外でも構わない。「@Idが付与されているプロパティが主キーに対応する」と解釈される。

④**@GeneratedValue**

・主キーが自動採番されることを表す。

・PostgreSQLでSERIAL型とした場合、strategyにはGenerationType.IDENTITYを指定する。

⑤**@Column**

・プロパティに対応するテーブルの列を指定する。

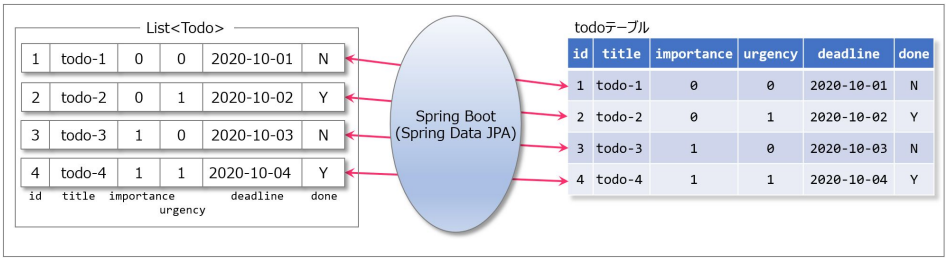
・プロパティ名と列名が同じなら省略可能(ただし大文字/小文字は区別されるので注意)。

・異なる名前にするときは指定が必要。

⇒テーブルのdone列をcheckedプロパティに対応付けるなら、以下のように付与する。

```
@Column(name = "done")
private String checked;
```

エンティティクラスとテーブルの関係を図で表すと次のようになります。



【図6-2】エンティティクラスとテーブルの対応

> @GeneratedValueのstrategyに指定する値は、使用するデータベースシステム、および自動採番の方法により異なります。興味がある方は調べてみてください。

6.4 リポジトリ

次はTodoRepository.javaです。

Spring Bootには、エンティティ(=テーブル)に対する処理を自動生成する仕組みがあります。これは「自分でコードを書かなくてよい」ということです。それを実現するのが、この**「リポジトリ(Repository)」**です。

【リスト6-2】com.example.todolist.repository.TodoRepository.java

```
package com.example.todolist.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.todolist.entity.Todo;

@Repository    //  ①
public interface TodoRepository extends JpaRepository<Todo, Integer>{
//  ②
}
```

TodoRepositoryインターフェースのポイントは、次の2か所です。

①**@Repository**

・このインターフェースがリポジトリであることを示すアノテーション。

・@Repositoryを付与したインターフェースの名称は、対象エンティティクラス名+"Repository"とするのが一般的。

②**JpaRepository<Todo, Integer>**

・インターフェースの継承元

・型引数<Todo, Integer>

第1引数：このリポジトリが対象とするエンティティ(クラス)

第2引数：対象エンティティで@Idが指定されているプロパティのクラス(Todoの場合、idなのでInteger)

このインターフェースには抽象メソッドがありません。しかし下表のメソッドが使用できるようになっています。テーブルに対するレコードの作成(Create)、読み出し

(**R**ead)、更新(**U**pdate)、削除(**D**elete)、いわゆるCRUDが一通り揃っていることがわかると思います。

【表6-5】自動実装されるメソッド(*:本書で使用するメソッド)

No	修飾子と型	メソッド(引数)	機能
1	long	count()	件数取得
2	<S extends T> long	count(Example<S> example)	
3	void	delete(T entity)	1レコード削除
4	void	deleteAll()	全レコード削除
5	void	deleteAll(Iterable<? extends T> entities)	該当レコード削除
6	void	deleteAllById(Iterable<? extends ID> ids)	主キーで削除
7	void	deleteAllByIdInBatch(Iterable<ID> ids)	
8	void	deleteAllInBatch()	全レコード削除
9	void	deleteAllInBatch(Iterable<T> entities)	該当レコード削除
10	void	**deleteById(ID id)***	主キーで削除
11	<S extends T>boolean	exists(Example<S> example)	存在チェック
12	boolean	existsById(ID id)	主

			キーで存在チェック
13	List<T>	**findAll()***	全レコード取得
14	<S extends T>Iterabe<S>	findAll(Example<S> example)	条件指定検索
15	Page<T>	**findAll(Pageable pageable)***	全レコード取得
16	List<T>	findAll(Sort sort)	
17	<S extends T>Page<S>	findAll(Example<S> example, Pageable pageable)	条件指定検索
18	<S extends T>List<S>	findAll(Example<S> example, Sort sort)	
19	List<T>	findAllById(Iterable<ID> ids)	
20	<S extends T, R>R	findBy(Example<S> example, Function<FluentQuery.FetchableFluentQuery<S>,R> queryFunction)	
21	Optional<T>	**findById(ID id)***	
22	<S extends T>Optional<S>	findOne(Example<S> example)	
23	void	flush()	更新結果をDBに反映
24	T	getReferenceById(ID id)	レコード

			への参照取得
25	<S extends T>S	save(S entity)	レコード追加・更新
26	<S extends T>Iterable<S>	saveAll(Iterable<S> entities)	
27	<S extends T>List<S>	saveAllAndFlush(Iterable<S> entities)	
28	<S extends T>S	**saveAndFlush(S entity)***	

T:エンティティの型，S:Tのサブクラス，ID:Tで@Idが指定されたプロパティの型
※deprecated(使用すべきではない)メソッドは上表に含めていません。

これらのメソッドはTodoRepositoryインターフェースの継承元であるJpaRepository、およびスパーインターフェースCrudRepository，PagingAndSortingRepository，QueryByExampleExecutorの抽象メソッドから継承されたものです。これはSpring Bootが自動実装します。

さらにTodoRepositoryインターフェースへ、ある「命名規則」に準じた名前の抽象メソッドを定義すると、それも自動実装してくれます(9章で説明します)。

ただしこの自動実装も、万能というわけではありません。複雑な検索や、動的に検索条件を組み立てるといった場合は、自分で検索メソッドを作成します(10章で説明します)。

6.5 コントローラー

TodoListControllerの役目は、GETリクエストを受け取ったらtodoテーブルを検索し、その結果を一覧画面(todoList.html)に渡すことです。検索には前節のTodoRepositoryを使います。

【リスト6-3】com.example.todolist.controller.TodoListController.java

```
package com.example.todolist.controller;

import java.util.List;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.servlet.ModelAndView;
import com.example.todolist.entity.Todo;
import com.example.todolist.repository.TodoRepository;
import lombok.AllArgsConstructor;

@Controller
@AllArgsConstructor                                    //  ②
public class TodoListController {
    private final TodoRepository todoRepository;      //  ①

    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv) {
        // 一覧を検索して表示する
        mv.setViewName("todoList");
        List<Todo> todoList = todoRepository.findAll(); //  ③
        mv.addObject("todoList", todoList);             //  ④
        return mv;
    }
}
```

前章までは、コントローラーのインスタンス変数に@Autowiredを付与していましたが、todoRepositoryには、ありません(①)。もし前章と同じように書くと、以下のようになり

ます。

```
  @Autowired
  private final TodoRepository todoRepository
```

これは「フィールドインジェクション」という方法です。以前はよく使われていました。しかし現在は、コンストラクタで初期化する「コンストラクタインジェクション」という方法が推奨されています。

```
  private final TodoRepository todoRepository
  @Autowired
  public TodoListController(TodoRepository todoRepository) {
    this.todoRepository = todoRepository;
  }
```

このように記述するとTodoListController起動時、Spring Bootが必要なインスタンスを生成し、コンストラクタを呼び出します。つまりtodoRepositoryは、Spring Bootがコンストラクタ経由でセットしてくれます。

> TodoRepository はインターフェースのためインスタンスを作れません。このときSpring BootはTodoRepositoryの実装クラスを無名クラスとして作成し、そのインスタンスをここで設定しています。

しかしこれだとフィールドインジェクションに比べ行数が増えます。そこでまず「コンストラクタが１つしかない場合、@Autowiredは省略できる」というSpring Bootのルールを利用します。

```
  private final TodoRepository todoRepository

  public TodoListController(TodoRepository todoRepository) {
    this.todoRepository = todoRepository;
  }
```

さらにこのコンストラクタは、クラスの(１つしかありませんが)全インスタンス変数に値をセットしています。つまりLombokの@AllArgsConstructor(②)を使えば、コンストラクタも省略できます。

```
  private final TodoRepository todoRepository      //  ①
```

以下本書では、この方法でコンストラクタインジェクションを使用します。

ハンドラーメソッドもシンプルです。http://localhost:8080/todoがリクエストされたら、showTodoList()を実行します。ポイントは次の2行です(③④)。

```
List<Todo> todoList = todoRepository.findAll(); //  ③
mv.addObject("todoList", todoList);             //  ④
```

③findAll()は【表6-5】にある自動実装されるメソッドの１つです。これはテーブルの全レコードを検索します。SELECT文で表せば「SELECT　*　FROM　todo」に相当します。

結果はList<Todo>型オブジェクトとして返されるので、そのまま一覧画面に渡して表示させます(④)。

6.6 ビュー

最後は検索したToDoの一覧表示画面です。

【リスト6-4】src/main/resources/templates/todoList.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>ToDo List</title>
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css"><!--
③ -->
</head>
<body>
  <table border="1">
    <tr>
      <th>id</th>
      <th>件名</th>
      <th>重要度</th>
      <th>緊急度</th>
      <th>期限</th>
      <th>完了</th>
    </tr>
    <tr th:each="todo:${todoList}"><!-- ① -->
      <td th:text="${todo.id}"></td>
      <td th:text="${todo.title}"></td>
      <td th:text="${todo.importance == 1 ? '★★★':'★'}"></td> <!-- ②
-->
      <td th:text="${todo.urgency == 1 ? '★★★':'★'}"></td>    <!-- ②
-->
      <td th:text="${todo.deadline}"></td>
      <td th:text="${todo.done == 'Y' ? '完了':''}"></td>        <!-- ②
-->
    </tr>
```

```
  </table>
</body>
</html>
```

検索結果の表示には、前章の数あてゲームと同じようにth:eachを使います(①)。

"todo:${todoList}"のtodoListは、前節のTodoListControllerからaddObject()で渡された、List<Todo>型のオブジェクトです。

変数todoは、このList<Todo>型の要素を表すのでTodo型です。そしてListの要素数分、つまり検索されたToDoの数だけ<tr>～</tr>を作成します。

todoに検索結果がセットされれば、あとはそれを表示するだけです。このうち「重要度」「緊急度」「完了」は、条件に応じて表示内容を変えています(②)。

【表6-6】「重要度」「緊急度」「完了」の表示内容

項目	条件	表示内容
重要度	todo.importance == 1	★★★
	上記以外	★
緊急度	todo.urgency == 1	★★★
	上記以外	★
完了	todo.done == 'Y'	完了
	上記以外	(何も表示しない)

この処理には条件演算子(三項演算子)を使っています。構文は以下の通りです。

```
条件式 ? 式1 : 式2
```

条件式がtrueなら式1を評価し、falseなら式2を評価します。

Javaのif文で表すと次のような意味です。

```
if ( 条件式 ){ 式1 } else { 式2 }
```

後述するようにThymeleafもif文相当の属性を持っていますが「true/falseで表示内容を変える」といった場合は、こちらの方がシンプルに書けます。

6.7 その他ファイル

ToDo一覧画面はCSSを利用して見栄えを整えています(③)。

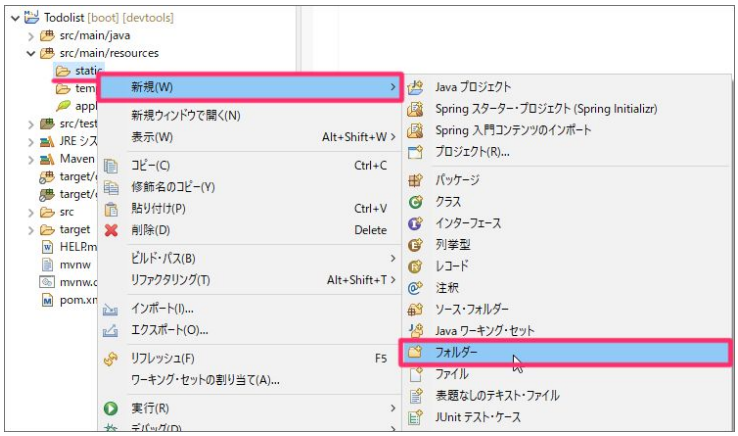
```
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css"><!--
③ -->
```

th:ref属性はHTMLのhref属性になります。また@{～}を**「リンクURL式」**と言います。

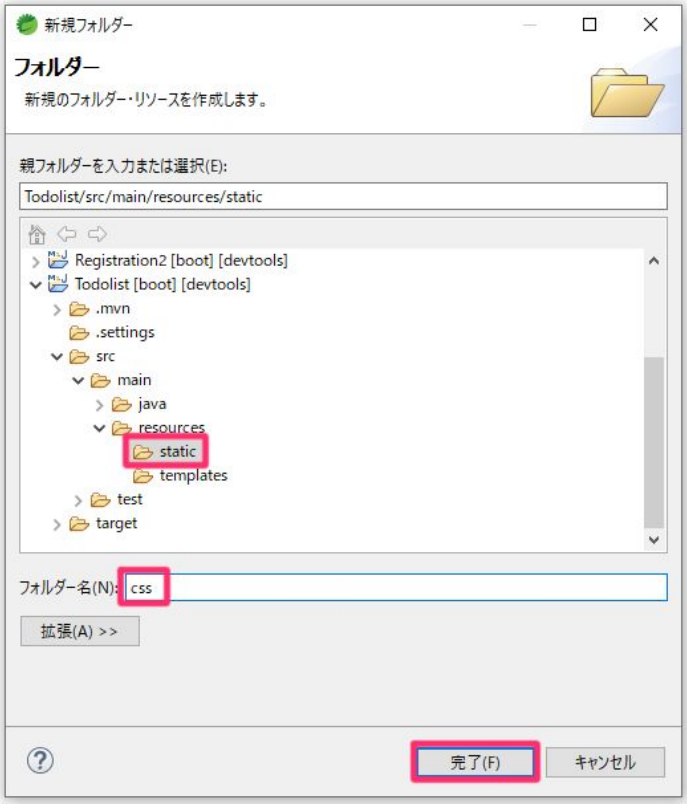
link要素のth:refにリンクURL式を使うと、staticフォルダを起点にしたパスと解釈されます。そのため、@{/css/style.css}は、static/css/style.cssを表します。

これに合わせてスタイルシートを配置する手順は、以下のようになります。

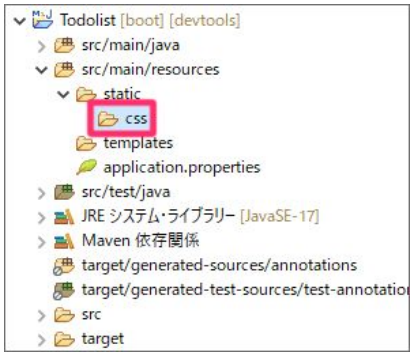
1)src/main/resources下のstaticを右クリック > [新規(W)] > [フォルダー]を選択する。



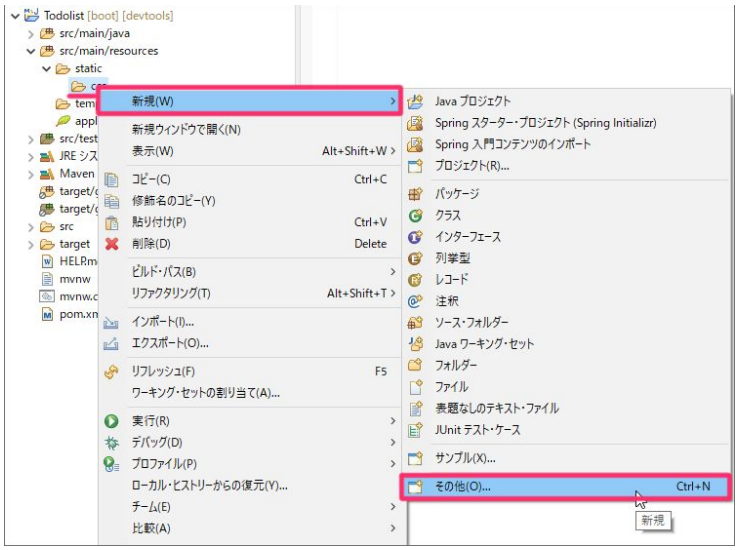
2)「新規フォルダー」ダイアログの[フォルダー名(N)]に"**css**"と入力 > [完了(F)]ボタンをクリックする。



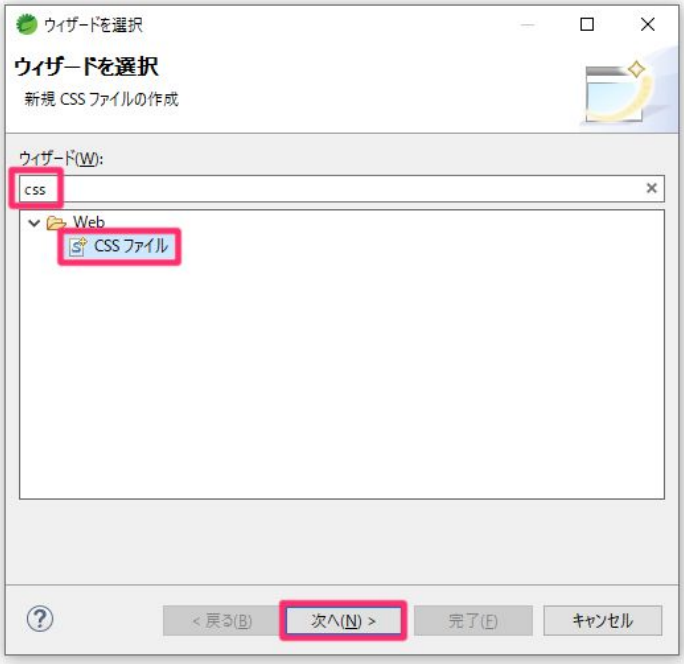
3)cssフォルダが作成される



4)cssフォルダを右クリック > [新規(W)] > [その他(O)...]を選択する。



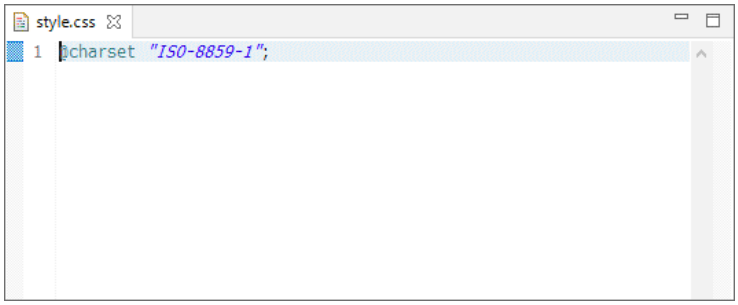
5)「新規」ダイアログの[ウィザード(W)]に"**css**"と入力 > [CSSファイル]をクリック > [次へ(N)>]ボタンをクリックする。



6)「新規CSSファイル」ダイアログでcssフォルダが選択されていることを確認し、[ファイル名(M)]に"**style.css**"入力>[次へ(N)>]ボタンをクリックする。

7)[完了(F)]ボタンをクリックする。

8)style.cssが作成される。



作成したstyle.cssを以下のように編集します。

【リスト6-5】src/main/resources/static/css/style.css

```
@charset "UTF-8";
body {
	margin: 20px;
	font-size: 14px;
}
table {
```

```
	border: solid 1px #84b2e0;
	border-collapse: collapse;
}
th {
	text-align: center;
	color: white;
	background: linear-gradient(#829ebc, #225588);
	box-shadow: 0px 1px 1px rgba(255, 255, 255, 0.3) inset;
	padding: 4px 10px;
}
td {
	padding: 4px 10px;
	border: solid 1px #84b2e0
}
button {
	width: 90px;
	margin: 10px 4px;
	padding: 4px;
}
.red {
	color:red;
}
#nav {
	list-style: none;
	display: flex;
	padding: 0px;
}

#nav li {
	margin-right: 10px;
}
```

ここまでの操作でsrc/main/resources下は、次のようになります。

```
Todolist
```

```
└ src/main/resources
   ├ static
   │  └ css
   │     └ style.css
   ├ templates
   │  └ todoList.html
   └ application.properties
```

最後はapplication.propertiesです。このファイルには、プロジェクト全般にかかわる情報を定義します。

本書ではPostgreSQLへの接続情報を記述しています。Spring Bootはプロジェクト起動時、このファイルを読み取り、PostgreSQLにアクセスできるよう準備します(PostgreSQLに接続するコードを書く必要はありません)。

パッケージ・エクスプローラーでapplication.propertiesをダブルクリックし、以下のように編集します。

【リスト6-6】src/main/resources/application.properties

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql:tododb
spring.datasource.username=todouser
spring.datasource.password=pass
```

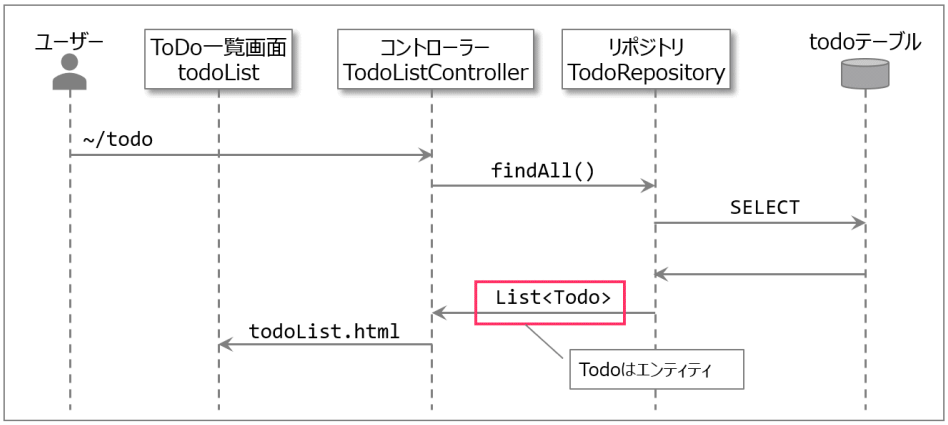
これは以下のような情報を設定しています。

```
spring.datasource.driver-class-name=<接続に使用するJDBCドライバークラス名>
spring.datasource.url=<JDBC URL>
spring.datasource.username=<接続に使用するユーザー名称>
spring.datasource.password=<接続に使用するユーザーのパスワード>
```

ここで作成したstyle.cssとapplication.propertiesは、これ以降の章でも使用します(内容は同じです)。

6.7 ビュー/コントローラー/リポジトリ/エンティティの関係

本章ではToDo一覧を表示するために必要なエンティティ、リポジトリ、コントローラー、そしてビュー(画面)を説明してきました。一度にたくさんの要素が出てきたので、少々混乱しているかもしれません。これらの関係をシーケンス図で表すと、以下のようになっています。



【図6-3】ToDo一覧表示のシーケンス

図中左上に～/todoとありますが、"～"は"http://localahost:8080"を略したものです。

この図と、本章のプログラムを照らし合わせながら、それぞれのつながりを把握できるようにしましょう。

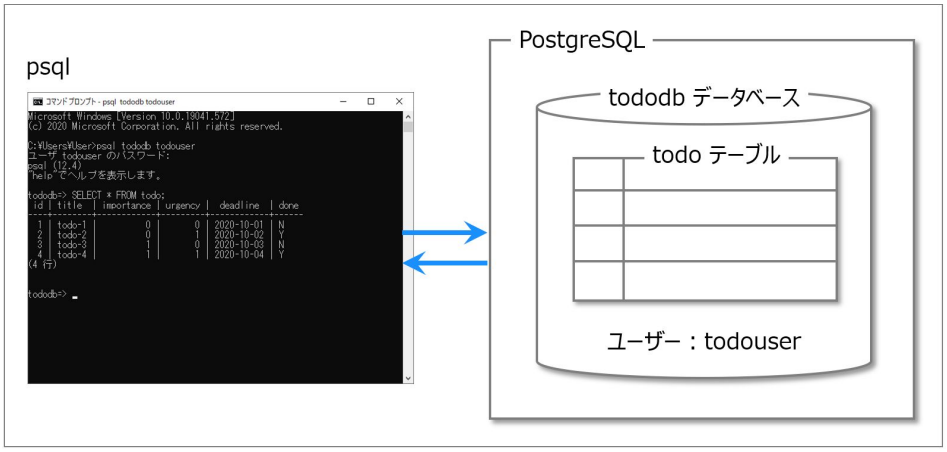
次章からは、ここで作成したプロジェクトをベースに各種機能を追加していきます。

todoテーブル作成手順

インストールしたPostgreSQL上に、下表の環境を作成します。

【表6-7】データベース環境

項目	名称	内容
データベース	tododb	todoテーブル格納する
ユーザー	todouser	tododbの所有者
パスワード	pass	todouserのパスワード
テーブル	todo	ToDoを格納する



【図6-4】作成するテーブルの構造

細かい説明は割愛しますが「データベースtododbの中にテーブルtodoがあり、todoにアクセスできるのはtodouserである」ということを覚えておいてください。

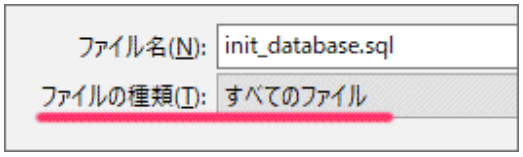
todoテーブル作成手順

1)任意のディレクトリに、以下のSQLファイルを、サクラエディタなどのエディタで作成する。

⇒Windows付属のメモ帳は、デフォルトで拡張子が“.txt”になるので、避けた方が無難

⇒どうしてもメモ帳で作成したい場合は、ファイルを格納するとき「名前を付けて保存」を使う。

そのときダイアログ下部の[ファイルの種類(T)]を「すべての種類」にし、ファイル名は～.sqlとする。



本書ではC:\temp下に作成したものとします。

【リスト6-7】C:\temp\init_database.sql

```
CREATE USER todouser WITH PASSWORD 'pass';
CREATE DATABASE tododb OWNER todouser ENCODING 'UTF8';
```

CREATE USER ～：ユーザー todouser を作成する。パスワードは pass とする。

CREATE DATABASE ～：データベース tododb を作成する。所有者は todouser 、文字セットは UTF8 とする。

【リスト6-8】C:\temp\init_table.sql

```
CREATE TABLE todo
(
  id           SERIAL PRIMARY KEY,
  title        TEXT,
  importance    INTEGER,
  urgency      INTEGER,
  deadline     DATE,
  done         TEXT
);

INSERT INTO todo(title, importance, urgency, deadline, done)
VALUES('todo-1', 0, 0, '2020-10-01', 'N');
INSERT INTO todo(title, importance, urgency, deadline, done)
VALUES('todo-2', 0, 1, '2020-10-02', 'Y');
INSERT INTO todo(title, importance, urgency, deadline, done)
VALUES('todo-3', 1, 0, '2020-10-03', 'N');
INSERT INTO todo(title, importance, urgency, deadline, done)
VALUES('todo-4', 1, 1, '2020-10-04', 'Y');
```

CREATE TABLE ～ : todoテーブルを作成する(【表6-1】参照)

INSERT ～ : todoテーブルにレコードを追加する。

INSERT文でテスト用のToDoを追加していますが、VALUE句にid列が含まれていないことに注意してください。
⇒id列はSERIAL型のため、自動的に1～の連番がセットされます。これは手順最後のSELECT文で確認します。

2)コマンドプロンプトを開く。
3)1)のSQLファイルを作成したディレクトリへ移動する。

```
>cd C:\temp
```

4)psql(PostgreSQLのコマンドラインベースの対話ツール)を起動する。
・接続ユーザー(-U)にはpostgresを指定する。
・パスワードはインストール時に、設定したものを入力する。

```
C:\temp>psql -U postgres
```

接続に成功するとプロンプトが以下のように変わります。

```
C:\temp>psql -U postgres
ユーザ postgres のパスワード:
psql (12.4)
"help"でヘルプを表示します。

postgres=#
```

5)1)のinit_database.sqlを実行する(\i に続けてSQLファイル名を指定する)
⇒これでデータベースとユーザーが作成される

```
postgres=# \i init_database.sql
CREATE ROLE
CREATE DATABASE
postgres=#
```

※もしメモ帳で作成して拡張子が“.txt”になった場合は、そのファイル名
(init_database.sql.txt)を入力する

6)\qを入力し、一度psqlを終了する。

```
postgres=# \q

C:\temp>
```

7)再びpsqlを起動して、tododbにtodouserとして接続する。パスワードはpassです。

```
C:\temp>psql tododb todouser
ユーザ todouser のパスワード:
psql (12.4)
"help"でヘルプを表示します。

tododb=>
```

8)2)init_table.sqlを実行する(\i に続けてSQLファイル名を指定する)
⇒これでテーブルが作成される

```
postgres=# \i init_table.sql
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
tododb=>
```

9)以下のSELECT文を実行してtodoテーブルにレコードが4件登録されていることを確認する。

```
SELECT  *  FROM  todo;
```

```
tododb=> SELECT * FROM todo;
 id | title  | importance | urgency |  deadline  | done
----+--------+------------+---------+------------+------
  1 | todo-1 |          0 |       0 | 2020-10-01 | N
  2 | todo-2 |          0 |       1 | 2020-10-02 | Y
  3 | todo-3 |          1 |       0 | 2020-10-03 | N
  4 | todo-4 |          1 |       1 | 2020-10-04 | Y
(4 行)


tododb=>
```

SERIAL型のid列に1～の連番が自動的にセットされています。

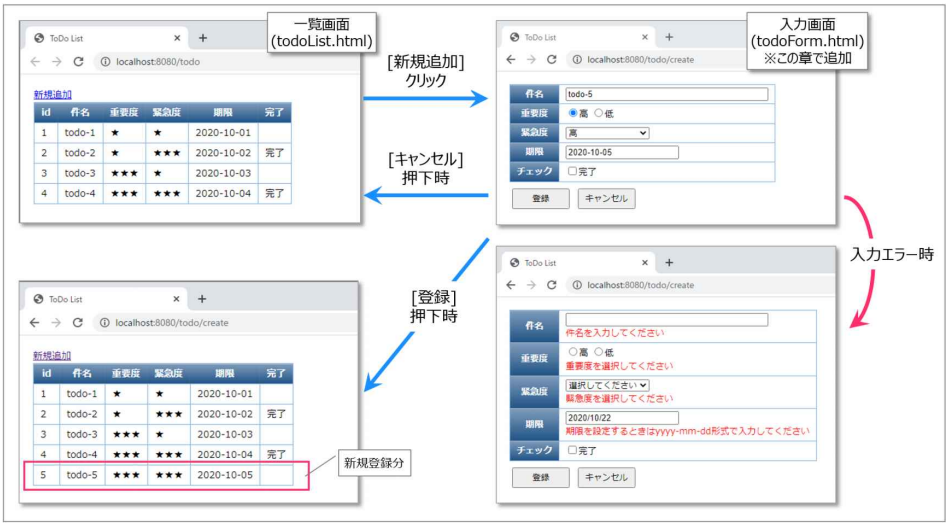
10)\qを入力してpsqlを終了する。

```
postgres=> \q


C:\temp>
```

11)コマンドプロンプトにexitと入力するか、右上の[X]をクリックして閉じる。

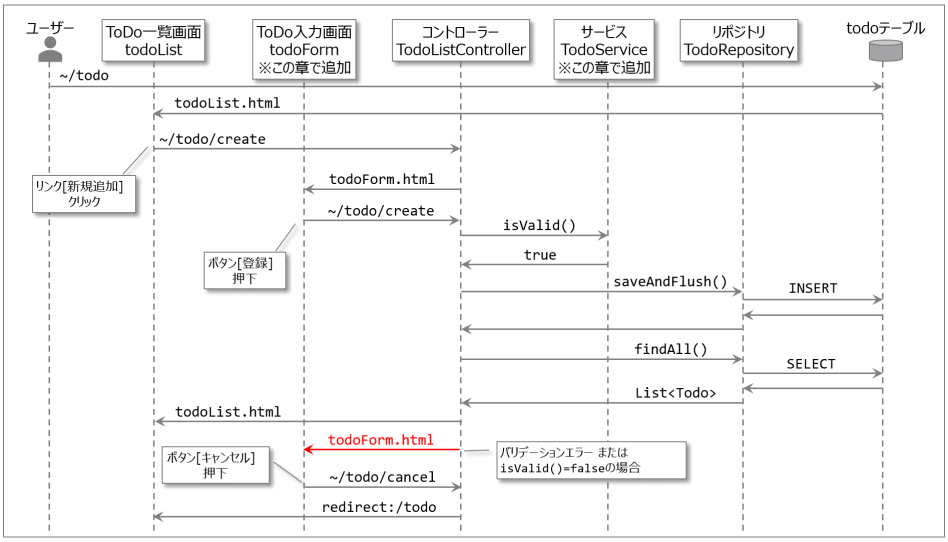
7．テーブルにレコードを追加する

本章ではToDoを追加できるようにします。入力画面だけでなく、入力データのチェック機能(バリデーション)も作成します。



【図7-1】ToDoの登録操作

シーケンス図で表すと、次のようになります。入力チェックは、アノテーションと新しく追加する「サービス」で行います。



※~/todoの詳細は6章参照(【図6-3】)

【図7-2】登録操作のシーケンス

作成するプロジェクトの仕様

プロジェクト名	Todolist2
パッケージ	com.example.todolist
依存関係	Spring Web, Spring Boot DevTools, Thymeleaf, Lombok, Spring Data JPA, PostgreSQL Driver, **Validation**
コントローラークラス	TodolistController
作成ファイル	src/main/resources/templates/todoForm.html(ToDo入力画面)

プロジェクトの構造は、次のようになります。

```
Todolist2
├ src/main/java
│  ├ com.example.todolist
│  │  └ Todolist2Application.java
│  ├ com.example.todolist.controller
│  │  └ TodoListController.java(▲)
│  ├ com.example.todolist.entity
│  │  └ Todo.java
│  ├ com.example.todolist.form(★)
│  │  └ TodoData.java(★)
│  ├ com.example.todolist.repository
│  │  └ TodoRepository.java
│  └ com.example.todolist.service(★)
│     └ TodoService.java(★)
└ src/main/resources
   ├ static
   │  └ css
   │    └ style.css
   └ templates
      ├ todoForm.html(★)
      └ todoList.html(▲)
```

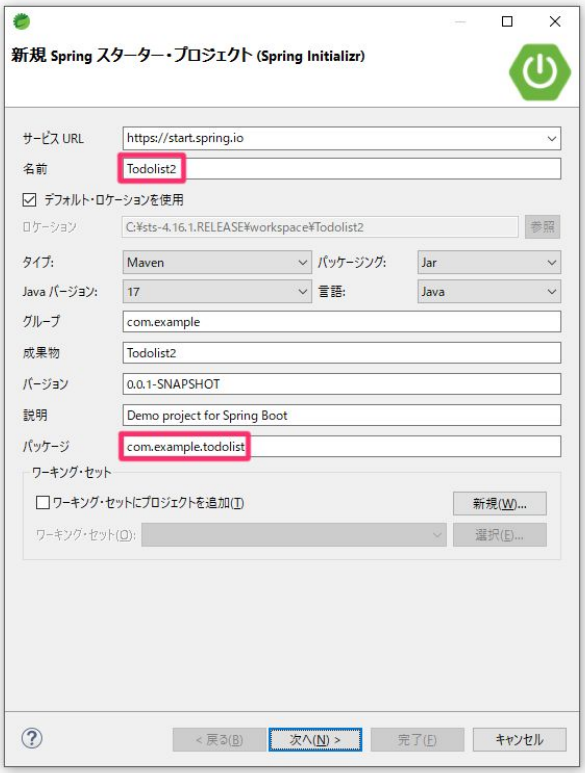
★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

本章からステップ・バイ・ステップで、前の章までの結果に追加、変更を加えていきます。説明は上図★▲の部分だけになります。

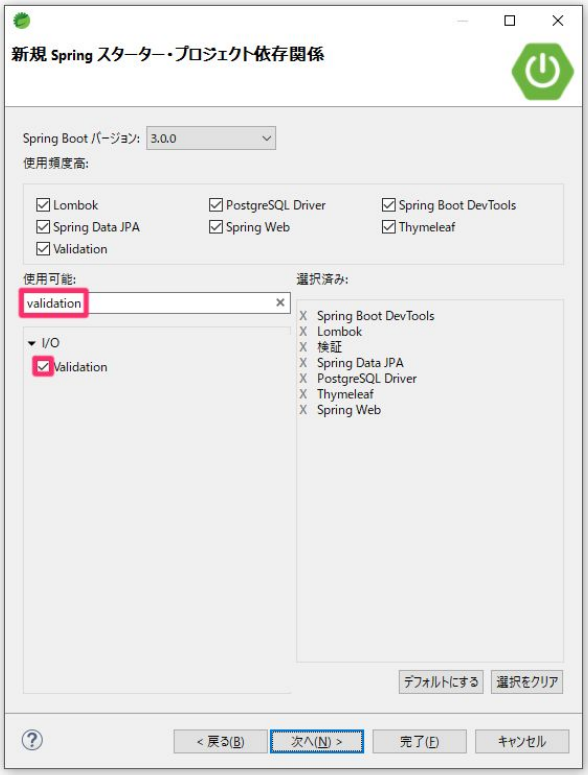
コードを入力しながら読み進まれる方は、プロジェクトのコピー方法を本章最後に載せていますので、そちらを参照し前章`Todolist`をコピー → 本章の内容を反映、とすることをお勧めします。

プロジェクト名は"**`Todolist2`**"にします。パッケージ名`com.example.todolist`、クラス名`TodoListController`は前章と同じです。



依存関係にバリデーション機能を追加します。[使用可能]に”**`validation`**”と入力し、それを選択してください。

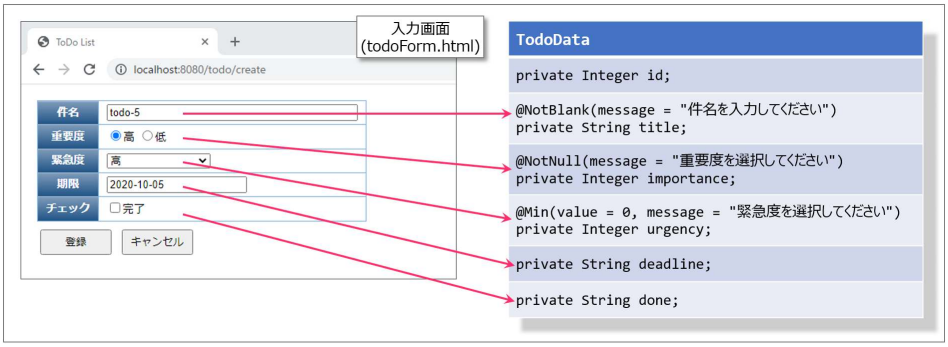
⇒ 選択済みには「**検証**」と日本語表示されます。


新規 Spring スターター・プロジェクト依存関係
Spring Boot バージョン:
3.0.0
使用頻度高:
Lombok
PostgreSQL Driver
Spring Boot DevTools
Spring Data JPA
Spring Web
Thymeleaf
Validation
使用可能:
validation
I/O
Validation
選択済み:
Spring Boot DevTools
Lombok
検証
Spring Data JPA
PostgreSQL Driver
Thymeleaf
Spring Web
デフォルトにする
選択をクリア
< 戻る(B)
次へ(N) >
完了(F)
キャンセル

7．1 バリデーション

最初にToDo入力画面(todoForm.html)のデータを取得するフォームクラスTodoDataを作成します。

フォームクラスの考え方は、4章で説明しました。フォーム部品と同じ名前のプロパティを定義するのがポイントです。このTodoDataも、後述のtodoForm.html(【リスト7-5】)と一致させています。これで@ModelAttributeにより、画面に入力された値を取得します。



【図7-3】フォームクラスTodoData

プロジェクトにパッケージcom.example.todolist.formを追加し、以下のクラスを定義します。

【リスト7-1】com.example.todolist.form.TodoData.java

```
package com.example.todolist.form;

import java.sql.Date;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import com.example.todolist.entity.Todo;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import lombok.Data;

@Data
public class TodoData {
```

```
    private Integer id;

    @NotBlank(message = "件名を入力してください")
    private String title;

    @NotNull(message = "重要度を選択してください")
    private Integer importance;

    @Min(value = 0, message = "緊急度を選択してください")
    private Integer urgency;

    private String deadline;
    private String done;

    /**
     * 入力データからEntityを生成して返す
     */
    public Todo toEntity() {
        Todo todo = new Todo();
        todo.setId(id);
        todo.setTitle(title);
        todo.setImportance(importance);
        todo.setUrgency(urgency);
        todo.setDone(done);

        SimpleDateFormat sdFormat = new SimpleDateFormat("yyyy-MM-dd");
        long ms;
        try {
            ms = sdFormat.parse(deadline).getTime();
            todo.setDeadline(new Date(ms));
        } catch (ParseException e) {
            todo.setDeadline(null);
        }
```

```
        return todo;
    }
}
```

TodoDataでは、アノテーションによる入力チェック(バリデーション;validation)を行います。

アノテーションとチェック内容は、以下のような関係になっています。

【表7-1】TodoDataクラスのバリデーション

プロパティ名	内容	アノテーション	チェック内容	エラーメッセージ(message)
title	件名	@NotBlank	空でないこと	"件名を入力してください"
importance	重要度	@NotNull	nullでないこと	"重要度を選択してください"
urgency	緊急度	@Min	指定値より大きいこと	"緊急度を選択してください"

各アノテーションは「チェック内容」が満たされていなければ、messageで指定されたエラーメッセージを生成します(チェックするタイミング、エラーメッセージの表示方法は後述します)。

Spring Bootは多くのバリデーションを持っていますが、大きく次の２つに分けられます。

1)Bean Validation

パッケージ名：jakarta.validation.constraints

(https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraints/package-summary.html)

2)Hibernate Validator

パッケージ名：org.hibernate.validator

(http://hibernate.org/validator/)

以下は代表的なバリデーションです。

【表7-2】Bean Validation(一部)

アノテーション	チェック内容	記述例
@NotNull	nullではないこと	@NotNull String title;
@NotEmpty	null,""でない	@NotEmpty String title;
@NotBlank	null,"",半角SPACE,TABでない	@NotBlank String title;
@Max	指定値以下であること	@Max(100) int score;
@Min	指定値以上であること	@Min(0) int score;
@Size	文字数/要素数が範囲内であること	@Size(min=1, max=20) String name;
@Future	現在日時より未来であること	@Future Date deadline;
@Past	現在日時より過去であること	@Past Date birthdate;
@AssertTrue	trueであること	@AssertTrue boolean isValid;
@AssertFalse	falseであること	@AssertFalse boolean isError;
@Pattern	正規表現にマッチすること	@Pattern(regexp="^[a-zA-Z]+$") String numeric;

【表7-3】Hibernate Validator(一部)

アノテーション	チェック内容	記述例
@Length	文字数が範囲内であること	@Length(min=0, max=100) String message;
@Range	数値が範囲内であること	@Range(min=0, max=100) int score;
@Email	Eメール形式であること	@Email String email;
@CreditCardNumber	クレジットカード番号形式であること	@CreditCardNumber String cardNumber;
@URL	URL形式であること	@URL String url;

> messageを指定しないと、デフォルトのエラーメッセージが使われます。(@NotBlankなら「空白は許可されていません」、など)。またプロパティファイルを用意すると、メッ

> セージの国際化に対応できます。興味がある方は調べてみてください。
> 本書の続編である「応用編」でも扱っています。

7．2 コントローラー

冒頭のシーケンス図を見ると、コントローラークラスには、新たに３つの処理が必要になることがわかります。

【処理1】ToDo一覧画面(todoList.html)で[新規追加]リンクがクリックされたときの処理 ⇒ 本節で説明

【処理2】ToDo入力画面(todoForm.html)で[登録]ボタンがクリックされたときの処理 ⇒ 本節で説明

【処理3】ToDo入力画面で[キャンセル]ボタンがクリックされたときの処理 ⇒ 7.5節で説明

これらを追加した本章のコントローラークラスは、次のようになっています。

【リスト7-2】com.example.todolist.controller.TodoListController.java

```
package com.example.todolist.controller;

import java.util.List;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.servlet.ModelAndView;
import com.example.todolist.entity.Todo;
import com.example.todolist.form.TodoData;
import com.example.todolist.repository.TodoRepository;
import com.example.todolist.service.TodoService;
import lombok.AllArgsConstructor;

@Controller
@AllArgsConstructor
public class TodoListController {
    private final TodoRepository todoRepository;
```

```
    private final TodoService todoService;  // Todolist2で追加

    // ToDo一覧表示(Todolistで追加)
    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv) {
        mv.setViewName("todoList");
        List<Todo> todoList = todoRepository.findAll();
        mv.addObject("todoList", todoList);
        return mv;
    }

    // ToDo入力フォーム表示(Todolist2で追加)
    // 【処理1】ToDo一覧画面(todoList.html)で[新規追加]リンクがクリックされたとき
    @GetMapping("/todo/create")
    public ModelAndView createTodo(ModelAndView mv) {
        mv.setViewName("todoForm");                 //  ①
        mv.addObject("todoData", new TodoData());   //  ②
        return mv;
    }

    // ToDo追加処理(Todolist2で追加)
    // 【処理2】ToDo入力画面(todoForm.html)で[登録]ボタンがクリックされたとき
    @PostMapping("/todo/create")
    public ModelAndView createTodo(@ModelAttribute @Validated TodoData todoData,//③
                                   BindingResult result,
                                   ModelAndView mv) {
      // エラーチェック
      boolean isValid = todoService.isValid(todoData, result); //  ④
      if (!result.hasErrors() && isValid) {
          // エラーなし
          Todo todo = todoData.toEntity();    //  ⑤
```

```
            todoRepository.saveAndFlush(todo);
            return showTodoList(mv);

        } else {
            // エラーあり
            mv.setViewName("todoForm");       // ⑥
            // mv.addObject("todoData", todoData);
            return mv;
        }
    }

    // ToDo一覧へ戻る(Todolist2で追加)
    // 【処理3】ToDo入力画面で[キャンセル登録]ボタンがクリックされたとき
    @PostMapping("/todo/cancel")
    public String cancel() {
        return "redirect:/todo";
    }

}
```

追加したハンドラーメソッドは、以下のようになっています(【処理3】[キャンセル]は7.5節参照)。

```
    @GetMapping("/todo/create")
    public ModelAndView createTodo(ModelAndView mv) {
        mv.setViewName("todoForm");                 //  ①
        mv.addObject("todoData", new TodoData());   //  ②
        return mv;
    }
```

【処理1】ToDo一覧画面の[新規追加]リンクに対応しています。
呼び出されたらToDo入力画面が表示されるようにします(①)。また前述のTodoDataオブジェクトを生成し、"todoData"という名称で画面に渡します(②)。これでToDo入力画面は、初期化された状態で始まります。

```
    @PostMapping("/todo/create")
```

```
  public ModelAndView createTodo(@ModelAttribute @Validated TodoData
todoData,//③
                                BindingResult result,
                                ModelAndView mv)
```

【処理2】のTodo入力画面の[新規登録]ボタンに対応しています。
ハンドラーメソッドの引数に、**@Validatedアノテーション**と**BindingResultオブジェクト**があります(③)。

@ValidatedはtodoDataにバインドされた値(=画面で入力された値)を、前節で説明したTodoDataクラスのアノテーション(@NotBlank,@NotNull,@Min)でチェックします。結果はBindingResultオブジェクトに格納されます。
つまりこのハンドラーメソッドが処理を始めるとき、resultにはtodoDataのチェック結果が入っています。
エラーがあればBindingResult#hasErrors()で判断します(エラーがあればtrueになる)。

さらにアノテーションでチェックできないエラーは、TodoService#isValid()で調べます(④)。詳細は次節で説明します。

```
  boolean isValid = todoService.isValid(todoData, result);//④
```

> ここで@GetMappingと@PostMappingのURLパスは同じ("/todo/create")ですが、リクエスト方法で区別されます。
> またハンドラーメソッドも、同じcreateTodoという名前です。これは引数の型・数が異なれば同名メソッドを定義できる「オーバーロード」を利用しています。

7．3 サービスクラス

アノテーションによる入力チェックは便利ですが、万能ではありません。たとえばToDo入力画面の件名に、全角スペースだけ入力しても、@NotBlankではエラーになりません。また期限の形式チェック(yyyy-mm-dd)も、標準のアノテーションでは難しいでしょう。

こういった場合は、自分で入力チェックを行い、結果をBindingResultに追加します。こうすればアノテーションと追加チェックの結果を1か所にまとめることができます。

追加チェックはコントローラークラスでも行えます。しかしSpring Bootで開発するときは、このような分岐のある複雑な処理はコントローラーに含めないようにするのが一般的です。代わりに別なクラスのメソッドとして定義し、そのクラスに**@Serviceアノテーション**を付与します。これを**「サービスクラス」**と言います。なおサービスクラスの名称は～Serviceとするのが一般的です。

以下が追加チェックを行うTodoServiceです。プロジェクトにパッケージcom.example.todolist.serviceを追加し、そこに定義します。

【リスト7-3】com.example.todolist.service.TodoService.java

```
package com.example.todolist.service;

import java.time.DateTimeException;
import java.time.LocalDate;
import org.springframework.stereotype.Service;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import com.example.todolist.form.TodoData;

@Service    // ①
public class TodoService {

    public boolean isValid(TodoData todoData, BindingResult result) {
        boolean ans = true;

        // 件名が全角スペースだけで構成されていたらエラー
```

```
String title = todoData.getTitle();
if (title != null && !title.equals("")) {
    boolean isAllDoubleSpace = true;
    for (int i = 0; i < title.length(); i++) {
        if (title.charAt(i) != '　') {
            isAllDoubleSpace = false;
            break;
        }
    }
    if (isAllDoubleSpace) {
        FieldError fieldError = new FieldError(
            result.getObjectName(),
            "title",
            "件名が全角スペースです");
        result.addError(fieldError);
        ans = false;
    }
}

// 期限が過去日付ならエラー
String deadline = todoData.getDeadline();
if (!deadline.equals("")) {
    LocalDate tody = LocalDate.now();
    LocalDate deadlineDate = null;
    try {
        deadlineDate = LocalDate.parse(deadline);
        if (deadlineDate.isBefore(tody)) {
            FieldError fieldError = new FieldError(
                result.getObjectName(),
                "deadline",
                "期限を設定するときは今日以降にしてください");
            result.addError(fieldError);
            ans = false;
         }
```

```
            } catch (DateTimeException e) {
              FieldError fieldError = new FieldError(
                  result.getObjectName(),
                  "deadline",
                  "期限を設定するときはyyyy-mm-dd形式で入力してください");
              result.addError(fieldError);
              ans = false;
            }
        }
        return ans;
    }
}
```

TodoServiceクラスには@Serviceを付与します(①)。前述のTodoListControllerは、このTodoServiceのインスタンスをコンストラクタインジェクションで取得しています。

ここでは追加のエラーチェックを行うisValid()だけ定義します。引数は画面入力値がバインドされたtodoDataとチェック結果を格納するresultです。todoDataをチェックし、エラーがあればFieldErrorオブジェクトをresultに追加(addError())します。

FieldErrorは以下のコンストラクターで生成しています(②)。引数はFormクラス名，エラーとするフィールド名，エラーメッセージの３つです。このうちFormクラス名はBindingResult#getObjectName()で取得できます。

```
  FieldError fieldError = new FieldError(  //  ②
                              result.getObjectName(),  //  Formクラス名
                              "title",                 //  フィールド名
                              "件名が全角スペースです");    //  エラーメッセージ
```

期限は、以下の2点をチェックします。これもエラーがあれば、FieldErrorをresultに追加します。

・LocalDate型に変換できる
・今日以降の日付である

> バリデーションを行うバリデーター(validator)を独自に作成することもできます。可能であれば、上記のような入力チェック処理もバリデーション化した方がよいでしょう。興味がある方は以下のインターフェース名などをキーに調べてみてください。
>
> `jakarta.validation.ConstraintValidator`

7．4 レコードの追加

【処理2】のハンドラーメソッドcreateTodo()へ戻ります。

```
boolean isValid = todoService.isValid(todoData, result);//  ④
if (!result.hasErrors() && isValid) {
    // エラーなし
    Todo todo = todoData.toEntity();     //  ⑤
    todoRepository.saveAndFlush(todo);
    return showTodoList(mv);

} else {
    // エラーあり
    mv.setViewName("todoForm");         //  ⑥
    // mv.addObject("todoData", todoData);
    return mv;
 }
```

アノテーションおよびisValid()でエラーがなければ、入力されたToDoをテーブルに追加します(⑤)。

追加はエンティティクラスのオブジェクトを引数にしてTodoRepository#saveAndFlush()を実行します(toEntity()はTodo型オブジェクトを返す。【リスト7-1】参照)。

この場合のsaveAndFlush()は、次のようなINSERT文に相当します。処理コードはSpring Bootが自動実装してくれるので、メソッドを実行するだけです(これも【表6-5】に含まれています)。

```
INSERT  INTO  todo(title,  importance,  urgency,  deadline,  done)
          VALUES(画面に入力された件名,  重要度,  緊急度,  期限,  完了)
```

ToDoを追加したら、ToDoリスト一覧を再表示します。ここでfindAll()を実行してもよいのですが、代わりに前章で作成したshowTodoList()を呼び出し、その戻り値をreturnします。これで一覧表示処理を1か所にまとめることができます(ただしこの方法には問題があるため、次章で解決します)。

```
return showTodoList(mv);
```

入力エラーのときは、入力画面へ戻り、エラーになった値を含め前回入力された値を再表示します。エラーメッセージもあわせて表示します(⑥)。

なおこの場合、入力画面にデータを渡すaddObject()は省略可能です。

```
mv.setViewName("todoForm");         //  ⑥
// mv.addObject("todoData", todoData); // ← 省略可
return mv;
```

これは「@ModelAttributeが付与されたオブジェクトは、addObject()しなくても遷移先で使用できる」からです。

ただし条件が１つあります。それはオブジェクトの名前(=addObject()の第1引数)が、「クラス名の先頭を小文字にしたもの」であることです。"todoData"はTodoData型オブジェクトtodoDataに対する名前なので、条件に合致します。よって省略可能です。

さらにバリデーション結果が格納されたBindingResultオブジェクトもaddObejct()しません。これはバリデーション対象オブジェクト(この場合todoData)に、関連付けられているからです。

7．5 リダイレクト

コントローラーの最後にある@PostMapping("/todo/cancel")は、【処理3】Todo入力画面の[キャンセル]ボタンに対応します。必要な機能はToDo一覧の再表示です。ここではshowTodoList()を直接実行するのではなく、**「リダイレクト(redirect)」**という方法を使い、次の画面へリクエストを転送します。

やり方は簡単で、ハンドラーメソッドの戻り値を"redirect:" + 転送先のURLパスとするだけです。

```
  @PostMapping("/todo/cancel")
  public String cancel() {
    return "redirect:/todo";
  }
```

これでURLパス/todoに対するGETリクエストが発生し、showTodoList()が呼び出されます。これがシーケンス図の一番下の往復になります。

> リクエストの転送には、この**リダイレクト(redirect)**と**フォワード(forward)**の2種類あります。興味がある方は違いなどについて調べてみてください。

7．5 エラーメッセージの表示

最後に画面を説明します。

前章で作成したToDo一覧画面には、入力画面へ遷移する[新規追加]リンクを追加します。これがコントローラーの【処理1】@GetMapping("/todo/create")と対応します。

【リスト7-4】src/main/resources/templates/todoList.html(一部抜粋)

```
<body>
<a th:href="@{/todo/create}">新規追加</a>
  <table>
```

本章からはa要素のURLパス指定にも、th:href属性とURLリンク式@{}を使います。

th:hrefはHTMLのhref属性になり、@{}はURLパスを生成します。ここでは単に<a href="/todo/create">としても同じですが、URLリンク式には「URLパスにパラメーターを埋め込む記述がシンプルになる」というメリットがあります(後述)。そのためURLリンク式で統一して行きます。

本章で追加するTodo入力画面は、次のようになっています。

【リスト7-5】src/main/resources/templates/todoForm.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>ToDo List</title>
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css">
</head>
<body>
  <form th:action="@{/}" method="post" th:object="${todoData}"><! -- ① -
->
    <table>
      <tr>
        <th>件名</th>
        <td>
          <input type="text" name="title" size="40" th:value="*{title}">
          <div th:if="${#fields.hasErrors('title')}" th:errors="*{title}"
```

```
            th:errorclass="red"></div>
        </td>
      </tr>
      <tr>
        <th>重要度</th>
        <td>
          <input type="radio" value="1" th:field="*{importance}">高
          <input type="radio" value="0" th:field="*{importance}">低
          <div th:if="${#fields.hasErrors('importance')}" th:errors="*
{importance}"
             th:errorclass="red"></div>
        </td>
      </tr>
      <tr>
        <th>緊急度</th>
        <td>
          <select name="urgency">
            <option value="-1" th:field="*{urgency}">選択してください
</option>
            <option value="1" th:field="*{urgency}">高</option>
            <option value="0" th:field="*{urgency}">低</option>
          </select>
          <div th:if="${#fields.hasErrors('urgency')}" th:errors="*
{urgency}"
             th:errorclass="red"></div>
        </td>
      </tr>
      <tr>
        <th>期限</th>
        <td>
          <input type="text" name="deadline" th:value="*{deadline}"
                 placeholder="yyyy-mm-dd">
          <div th:if="${#fields.hasErrors('deadline')}" th:errors="*
{deadline}"
```

```
            th:errorclass="red"></div>
        </td>
      </tr>
      <tr>
        <th>チェック</th>
        <td>
          <input type="checkbox" value="Y" th:field="*{done}">完了
          <input type="hidden" name="!done" value="N" />
        </td>
      </tr>
    </table>
    <div>
      <button type="submit" th:formaction="@{/todo/create}">登録</button>
      <button type="submit" th:formaction="@{/todo/cancel}">キャンセル
</button>
    </div>
  </form>
</body>
</html>
```

一番のポイントはform要素のth:object="${todoData}"です(①)。

```
  <form th:action="@{/}" method="post" th:object="${todoData}"><! -- ① -
->
```

th:object属性は、その要素内(下位要素)で使用するオブジェクトを設定します。このオブジェクトからは*{プロパティ名}で、プロパティ値を取得できます(*{}を**選択変数式**と言います)。

ここではform要素にth:objectがあるので、下位のフォーム部品でtodoDataが使えます。

*{プロパティ名}は、件名の**th:value**属性などで使われています。

```
  <input type="text" name="title" size="40" th:value="*{title}">
```

```
    ↓
  <input type="text" name="title" size="40" value="todoData.getTitle()の結果">
```

まずth:valueはHTMLのvalue属性を作成します。値の*{title}は「th:objectで指定されたtodoDataのtitleプロパティの値」と解釈され、todoData.getTitle()の結果がvalue属性の値になります。

次のラジオボタン(type="radio")には**th:field属性**があります。

```
  <input type="radio" value="1" th:field="*{importance}">高
```

th:fieldはHTMLのid属性，name属性，value属性を生成します。

```
  id="importance", name="importance", value="todoData.getImportance()の結果"
```

しかしvalue属性は最初から記述されているので(value="1")、こちらが使われます。

```
  id="importance", name="importance", value="1"
```

さらにこのラジオボタンには、同じ名前の要素が複数あります。こういった場合、id属性には1～の連番が追加されます。

よって次のようになります。

```
  <input type="radio" value="1" th:field="*{importance}">高
  <input type="radio" value="0" th:field="*{importance}">低
    ↓
  <input type="radio" value="1" id="importance1" name="importance">高
  <input type="radio" value="0" id="importance2" name="importance">低
```

th:fieldはoption要素でも使っています。

```
  <option value="-1" th:field="*{urgency}">選択してください</option>
```

```
  <option value="1" th:field="*{urgency}">高</option>
  <option value="0" th:field="*{urgency}">低</option>
```

上記のようにth:fieldはid,name,value属性を作成しますが、option要素はid, name属性を持ちません。またvalue属性も明示してあるので、その値になります。よってth:fieldを指定する意味が無いように見えます。

しかしth:fieldには「option要素のvalue属性の値=*{プロパティ名}の値なら、selected属性を追加する」という働きがあります。もしtodoData.urgency=1なら、次のようなHTMLが生成されます。

```
  <option value="-1">選択してください</option>
  <option value="1" selected>高</option>
  option value="0">低</option>
```

これはラジオボタン、チェックボックスのchecked属性に対しても同様です。値が同じならchecked属性が追加されます。

完了のチェックボックスには、hidden要素も使っています。

```
  <input type="checkbox" value="Y" th:field="*{done}">完了
  <input type="hidden" name="!done" value="N" />
```

チェックボックスの値がサーバーへ送信されるのは、チェックされているときだけです。未チェックのまま、あるいはチェックから未チェックに変更した場合、サーバーには何も送られません。そこで上記hidden要素を追加することで、未チェック時の値もフォームオブジェクトにバインドできるようにします。

このhidden要素のname属性は、対応するチェックボックスの名前の先頭に!をつけたものです。

value属性は、未チェック時の値です。

これで未チェック時、todoData.doneには“N”がセットされます。

> チェックボックスの値がboolean(true/false)なら、上記hidden要素の追加は不要です。

[登録]ボタン(【処理2】に対応)、[キャンセル]ボタンは<input type="submit">ではなく、汎用のbutton要素としています。

```
<button type="submit" th:formaction="@{/todo/create}">登録</button>
```

```
<button type="submit" th:formaction="@{/todo/cancel}">キャンセル</button>
```

これは押されたボタンによってURLパスを変えたいためです。それぞれのURLパスは**th:formaction属性**で指定します。このth:formactionはHTMLのformaction属性となり、form要素のaction属性を上書きします(【表4-2】参照)。

最後はエラーメッセージの表示です。以下は件名(=title)に対するものですが、少々複雑です。

```
<div th:if="${#fields.hasErrors('title')}" th:errors="*{title}"
            th:errorclass="red"></div>
```

このdiv要素は、件名にエラーがあった場合のみエラーメッセージを表示するために作成します。

・**#fields.hasErrors('title')**で、エラー有無をチェックします。
#fieldsはth:objectで指定したオブジェクトに関連付けられたエラー情報を表します。ここでは、ハンドラーメソッドcreateTodo()のresultになります。中にはアノテーションとTodoService#isValid()のチェック結果が格納されています。そこにtitleのエラーがあればtrue、なければfalseが返されます。

・**th:if属性**は、この要素を作成するかどうかの判断です。
変数式がtrueならこの要素が作成される、falseなら作成されません。

・**th:errors属性**は、エラーメッセージを取得します。
*{プロパティ名}の形式で、どのプロパティに対するエラーメッセージなのかを指定します。

・**th:errorclass属性**は、この要素に適用するCSSのクラス名称です。
redはstyle.cssで以下のように定義しているので、エラーメッセージは赤文字で表示されます。

```
.red {
      color:red;
}
```

注意点としてth:objectとエラー表示要素の位置関係があります。

このtodoForm.htmlでは、エラー表示するdiv要素を、th:objectを指定したform要素の内部(下位)要素としています。

```
<form th:action="@{/}" method="post" th:object="${todoData}">
      :
  <div th:if="${#fields.hasErrors('title')}" th:errors="*{title}"
th:errorclass="red"></div>
      :
</form>
```

これを以下のように上位-下位関係の無いところに配置すると、Thymeleafでエラーになるので注意してください。

```
<form th:action="@{/}" method="post" th:object="${todoData}">
      :
</form>
<div th:if="${#fields.hasErrors('title')}" th:errors="*{title}"
th:errorclass="red"></div>
```

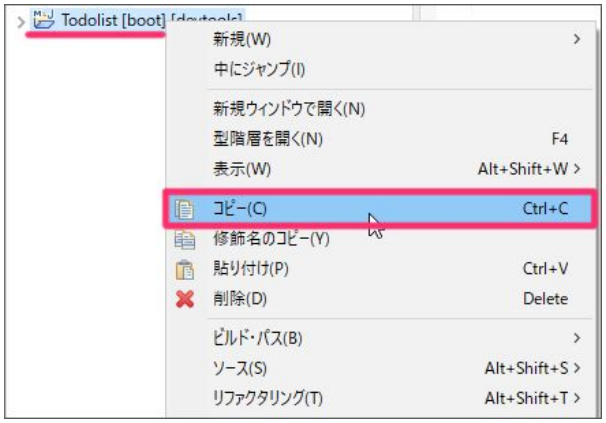
> 次のようにするとエラーの一覧表示もできます。ただし、エラーを発生箇所から離れたところに、まとめて表示するというのは、ユーザーインターフェース的に良い方法とは言えないでしょう。
>
> ```
> <form th:action="@{/}" method="post" th:object="${todoData}">
> <ul>
>   <li th:each="e : ${#fields.detailedErrors()}" th:text="${e.message}"
> class="red"></li>
> </ul>
> ```

本章では新たに入力画面を追加し、ToDoを登録できるようにしました。次章からは、ToDoの変更などができるようにしていきます。

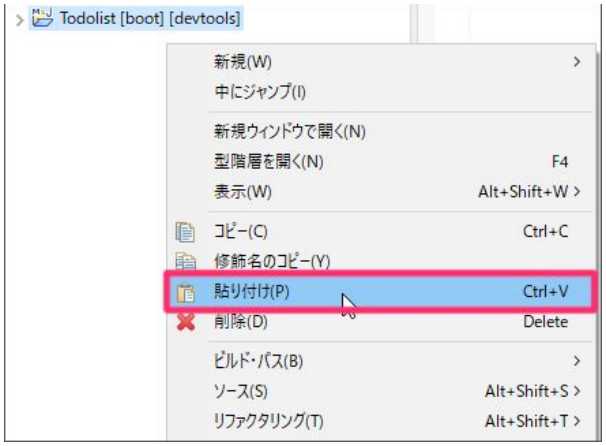
補足：プロジェクトのコピー方法

本節ではTodolistプロジェクトをコピーしてTodolist2を作成する手順を説明します。これ以降の章でも、同様の手順で前の章のプロジェクトをコピーできます。

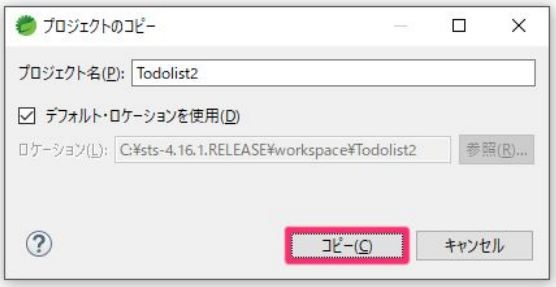
1)パッケージ・エクスプローラーでコピー元のプロジェクト(Todolist)を右クリック > [コピー(C)]を選択する。



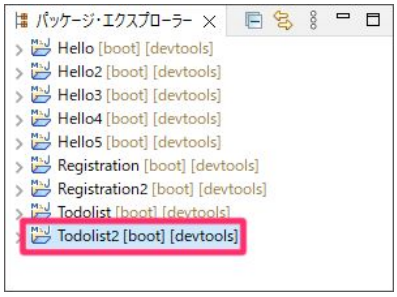
2)パッケージ・エクスプローラーの下の余白を右クリック > [貼り付け(P)]を選択する。



3)「プロジェクトのコピー」ダイアログの[プロジェクト名(P)]にコピー先のプロジェクト名(**"Todolist2"**)を入力 > [コピー(C)]ボタンをクリックする。



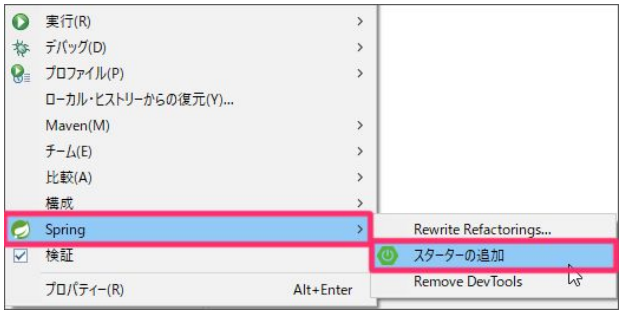
4)プロジェクトがコピーされる(Todolist2)。



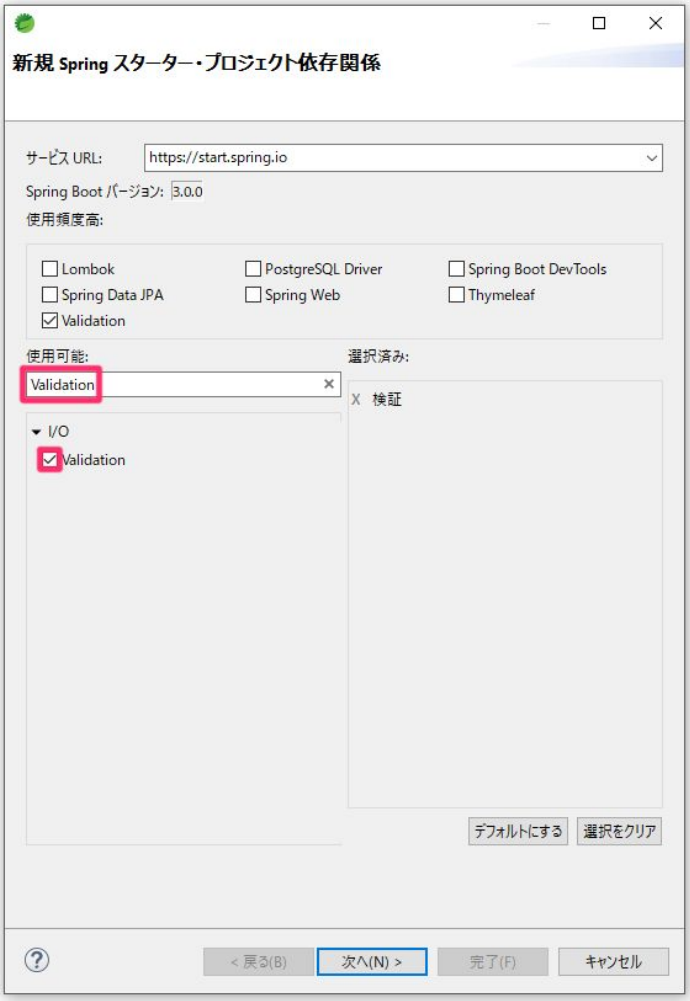
本章から新たに依存関係として「Validation」を使いますので、コピーしたプロジェクトに以下の手順で追加します。

なお本章以降、依存関係は同じです。よってこの操作が必要なのは、Todolist→Todolist2の場合だけです。

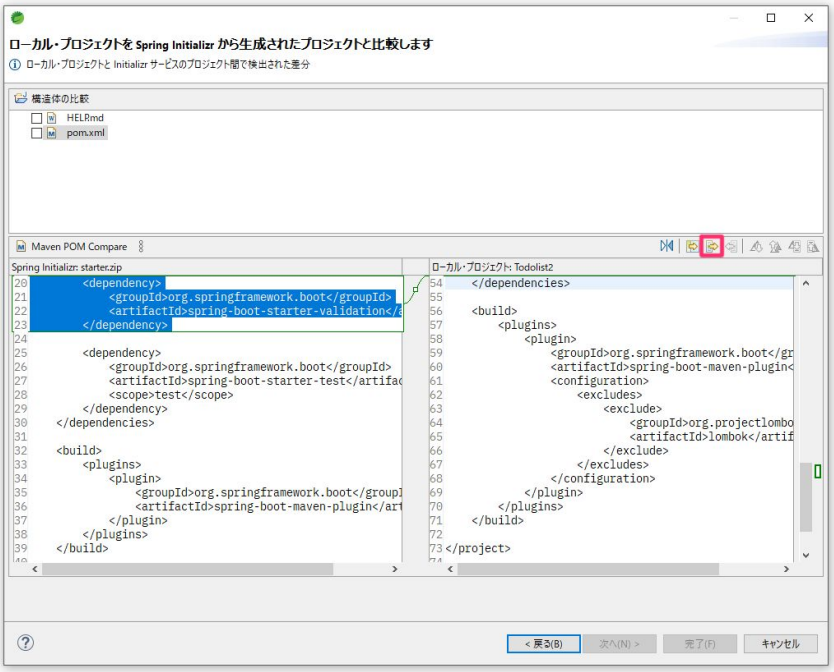
5)コピーしたプロジェクト(Todolist2)を右クリック > [Spring] > [スターターの追加(S)]を選択する。



6)「新規Springスターター・プロジェクト依存関係」ダイアログが表示されるので、[使用可能]に"**Validation**"と入力。チェック > [次へ(N)>]ボタンをクリックする。

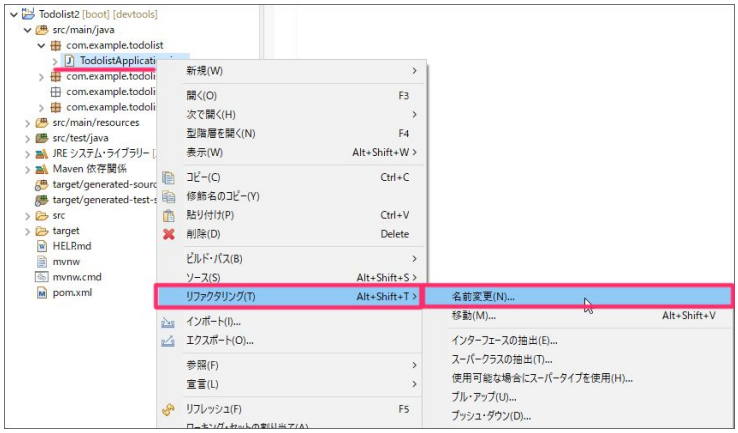


7)変更箇所が表示されるので[左から右へ現在の変更をコピーする]ボタンをクリック ›
[完了(F)]ボタンをクリックする。



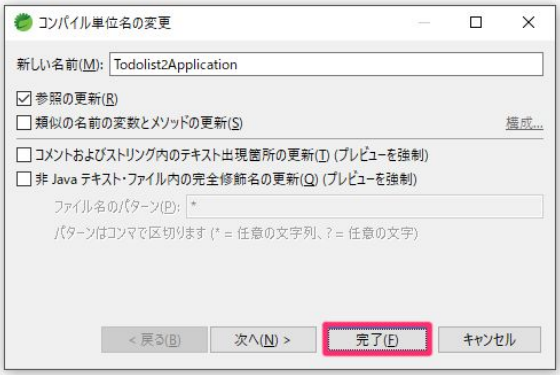
ここまでの操作で本書の内容を進めていく準備ができました。ただし既存のプロジェクトを直接コピーして作ったため、一部不整合が起きています。ToDoアプリケーションを動かすのには支障ありませんが、念のため以下の手順で解消します。

8)コピーしたプロジェクトのパッケージcom.example.todolist下のTodolistApplication.javaを右クリック > [リファクタリング(T)]>[名前変更(N)...]を選択する。

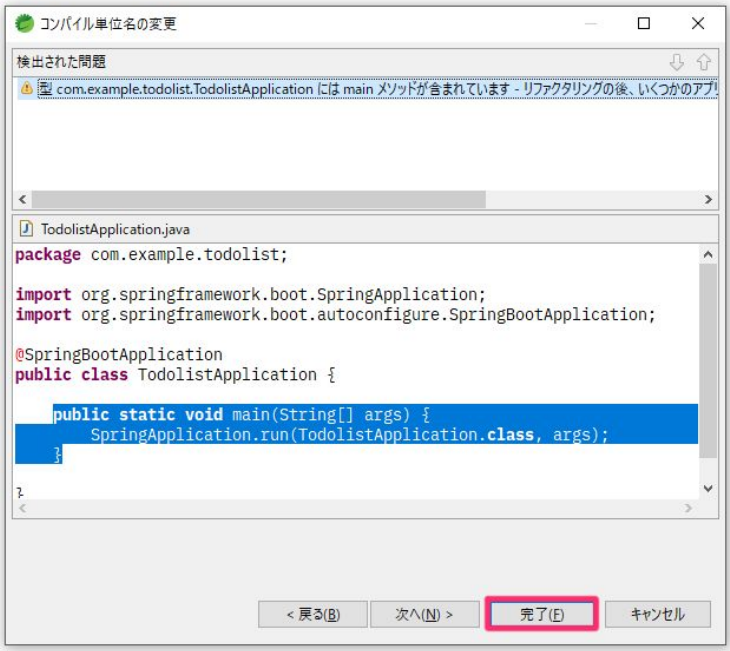


9)「コンパイル単位名の変更」ダイアログで、[新しい名前(M)]にコピー先プロジェクト名+"Application("**Todolist2Application**")を入力 > [次へ(N)>]ボタンをクリック

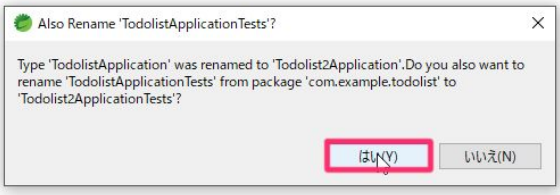
する。



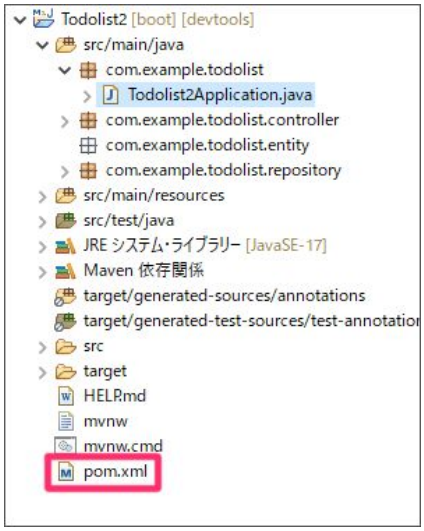
10)[完了(F)>]ボタンをクリックする。



11)[はい(Y)>]ボタンをクリックする。



12)コピー先プロジェクト(Todolist2)下にあるpom.xmlをダブルクリックして開く。



13)先頭付近の<artifactId>～</artifactId>と<name></name>の2か所を、コピー先プロジェクト名("**Todolist2**")に変更し、保存する。

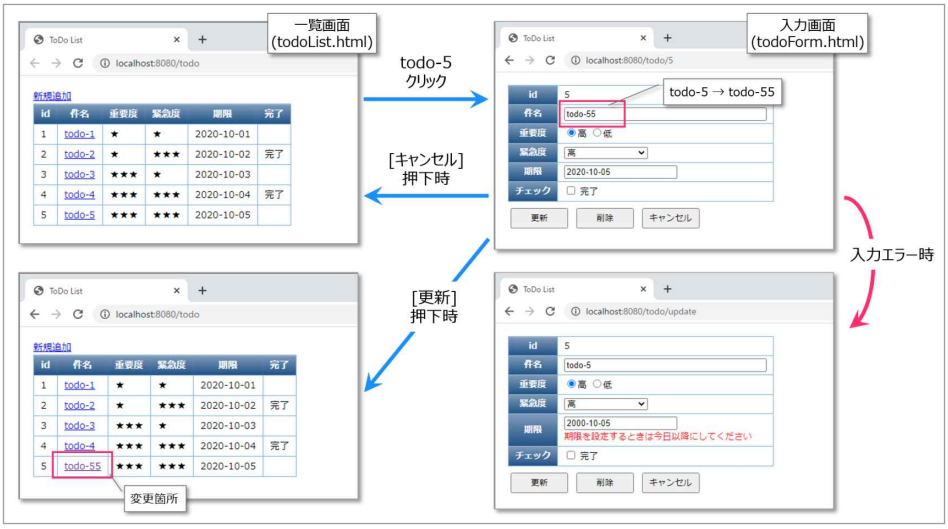
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"...(略)
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 ...(略)
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
        artifactId>Todolist2</artifactId>  <!-- ←変更 -->
    <version>0.0.1-SNAPSHOT</version>
    <name>Todolist2</name>  <!-- ←変更 -->
    <description>Demo project for Spring Boot</description>
```

> pom.xmlはMaven(メイヴン/メイヴェン)というJavaプロジェクト用プロジェクト管理ツールの定義ファイルです。このファイルで、作成するプロジェクトの情報や依存関係などを管理しています。

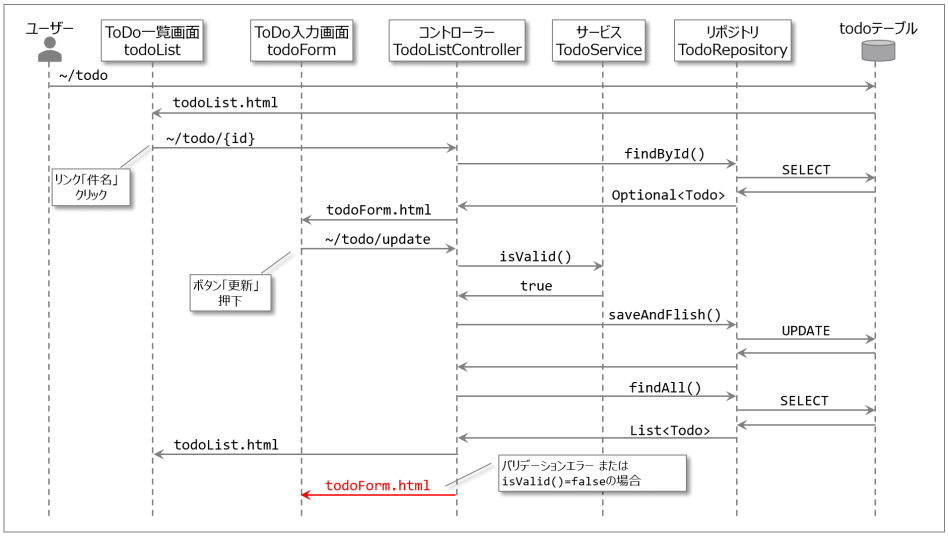
> STSではGrandleという管理ツールも使えますが、デフォルトはMavenです(「新規Springスターター・プロジェクト」ダイアログの[タイプ]で指定している)。MavenやGrandleはSTS以外でも利用されています。興味がある方は調べてみてください。

8．テーブルのレコードを更新・削除する

前章までででToDoの一覧表示、追加ができるようになりました。本章ではさらに更新機能と削除機能を追加します。また追加後の画面再表示についても見直します。

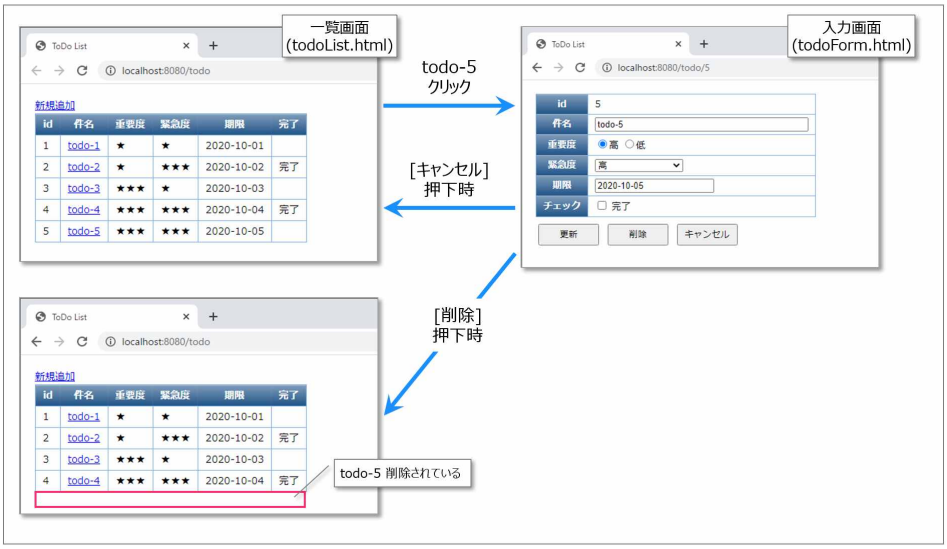


【図8-1】ToDoの更新操作

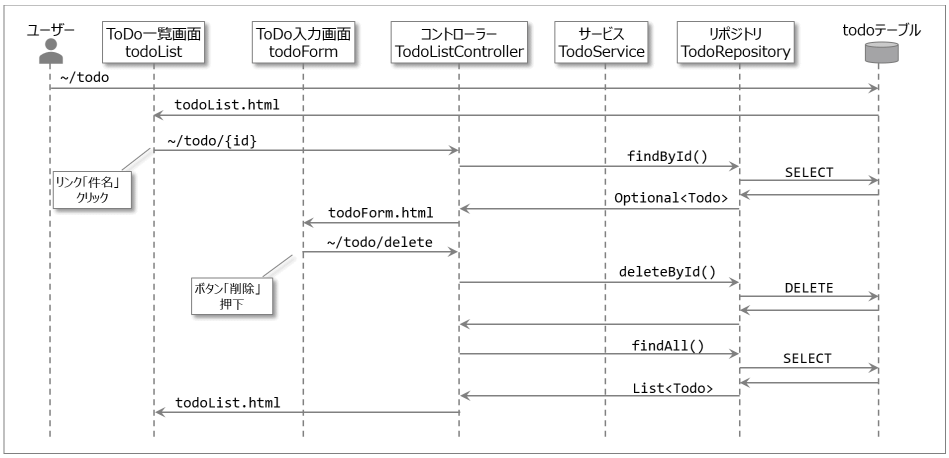


※~/todoの詳細は6章参照(【図6-3】)

【図8-2】更新操作のシーケンス



【図8-3】ToDoの削除操作



※~/todoの詳細は6章参照(【図6-3】)

【図8-4】削除操作のシーケンス

作成するプロジェクトの仕様

プロジェクト名	Todolist3
依存関係	Spring Web, Spring Boot DevTools, Thymeleaf, Lombok, Spring Data JPA, PostgreSQL Driver, Validation

```
Todolist3
```

```
├ src/main/java
│  ├ com.example.todolist
│  │  └ TodolistApplication.java
│  ├ com.example.todolist.controller
│  │  └ TodoListController.java(▲)
│  ├ com.example.todolist.entity
│  │  └ Todo.java
│  ├ com.example.todolist.form
│  │  └ TodoData.java
│  ├ com.example.todolist.repository
│  │  └ TodoRepository.java
│  └ com.example.todolist.service
│     └ TodoService.java
└ src/main/resources
   ├ static
   │  └ css
   │     └ style.css
   └ templates
      ├ todoForm.html(▲)
      └ todoList.html(▲)
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

8.1 主キーで検索する

更新と削除は、一覧画面(todoList.html)から対象とするToDoを選ぶことから始めます。

一覧画面の件名がリンク(a要素)になっていて、クリックすると入力画面(todoForm.html)へ遷移し、[更新][削除]ボタンをクリックできます(【図8-1】,【図8-3】参照)。

この件名リンクには、選択されたToDoをコントローラーに知らせる情報が必要です。そこでtodoテーブルの主キーであるidの値を含めます。具体的にはURLパスを「/todo/todoのid」とし、idを@PathVariableで受け取ります。

【リスト8-1】src/main/resources/templates/todoList.html(変更箇所)

```
<td th:text="${todo.title}"></td>
↓
<td><a th:href="@{/todo/__${todo.id}__}" th:text="${todo.title}"></a>
</td>
```

URLリンク式(@{})の中で、todo.idを囲んでいる__${}__(変数式の前後にアンダースコアが２つ)は、値を埋め込むための書き方です。これでtodoオブジェクトがid=1, title="todo-1"なら、次のようなHTMLになります。

```
<td><a href="/todo/1">todo-1</a></td>
```

このリクエストを受け取るコントローラーには、以下のハンドラーメソッドを追加します。

【リスト8-2】

com.example.todolist.controller.TodoListController.java#todoById()

```
  private final HttpSession session;
        :
  @GetMapping("/todo/{id}")
  public ModelAndView todoById(@PathVariable(name = "id") int id,
ModelAndView mv) {
```

```
        mv.setViewName("todoForm");
        Todo todo = todoRepository.findById(id).get();   //  ①
        mv.addObject("todoData", todo);                    //  ※b
        session.setAttribute("mode", "update");          //  ②
        return mv;
    }
```

URIテンプレート変数idを@PathVariableで取得し、それを引数にfindById()でtodoテーブルを検索します(①)。findById()は「@Idが付与された項目を条件に検索」するメソッドで、【表6-5】に含まれています。

SELECT文で表せば次のようになります。

```
SELECT  *  FROM  todo  WHERE  id  =  引数idの値
```

①ではさらにfindById()の戻り値にget()を実行します。これはfindById()がOptional<Todo>型オブジェクトを返すためです(【表6-5】参照)。

findById()は@Id(主キー)で検索するため、結果は1件または0件です。該当があれば、そのレコードを表すTodoオブジェクト、なければnullという意味のOptional<Todo>です。get()はこのOptionalオブジェクトからTodoオブジェクトを取得するものです(java.util.Optional#get())。

> 実務ではfindById()がnullの場合も考慮すべきですが、本書では省略しています。リンクがクリックできるということは、showTodoList()でfindAll()を実行したときには存在していたはずです。つまり一覧が表示されから、リンクをクリックするまでの間にレコードが削除されない限りnullにはなりません。

検索結果todoはaddObjectメソッド(※b)で入力画面に渡します。

また入力画面から更新、削除もできるようにするため、表示するボタンは機能にあわせて変える必要があります。そこでコンストラクタインジェクションで取得したセッション(HttpSession型の変数session)に、入力画面のボタンを指定するデータを書き込んでおきます(②)。入力画面はこの内容を見て、表示するボタンを切り替えます(後述)。

【表8-1】表示するボタンの切り替え

イベント	入力画面に表示するボタン	セッションに格納するデータ

新規追加をクリック	[登録][キャンセル]	mode="create"
件名をクリック	[更新][削除][キャンセル]	mode="update"

mode="create"も、createTodo()で書き込むようコードを追加します(③)。

【リスト8-3】

com.example.todolist.controller.TodoListController.java#createTodo()

```
  @GetMapping("/todo/create")
  public ModelAndView createTodo(ModelAndView mv) {
      mv.setViewName("todoForm");
      mv.addObject("todoData", new TodoData());  //  ※a
      session.setAttribute("mode", "create");    //  ③
      return mv;
  }
```

これで件名をクリックしたToDoの内容が入力画面に表示されます。

8.2 レコードを更新する

次は入力画面で[更新]ボタンが押された時の処理です。ハンドラーメソッドは以下のようになっています。

【リスト8-4】

com.example.todolist.controller.TodoListController.java#updateTodo()

```
import org.springframework.ui.Model;
    :
  @PostMapping("/todo/update")
  public String updateTodo(@ModelAttribute @Validated TodoData todoData,
                           BindingResult result,
                           Model model) {
      // エラーチェック
      boolean isValid = todoService.isValid(todoData, result);
      if (!result.hasErrors() && isValid) {
          // エラーなし
          Todo todo = todoData.toEntity();
          todoRepository.saveAndFlush(todo);   //  ①
          return "redirect:/todo";

      } else {
          // エラーあり
          // model.addAttribute("todoData", todoData);
          return "todoForm";
      }
  }
```

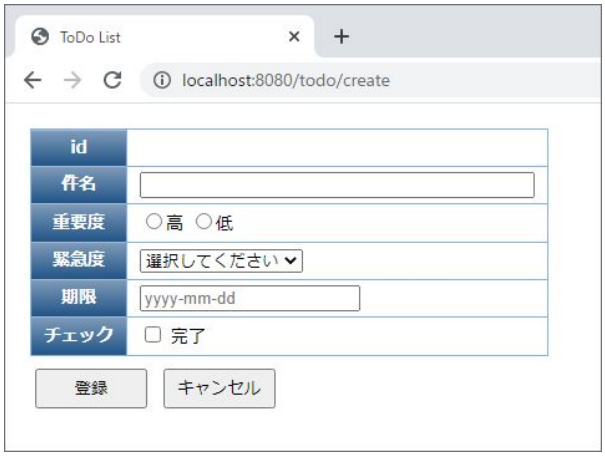
前章の新規登録createTodo()(【リスト7-2】)と、よく似ています。これはエラーチェック方法が同じで、さらに更新もsaveAndFlush()で行うためです(①)。

saveAndFlush()は、引数のエンティティオブジェクトで、@Idが付与されたプロパティがnullならレコードを追加します(前章の内容)。nullでなければ、その値を条件にして引

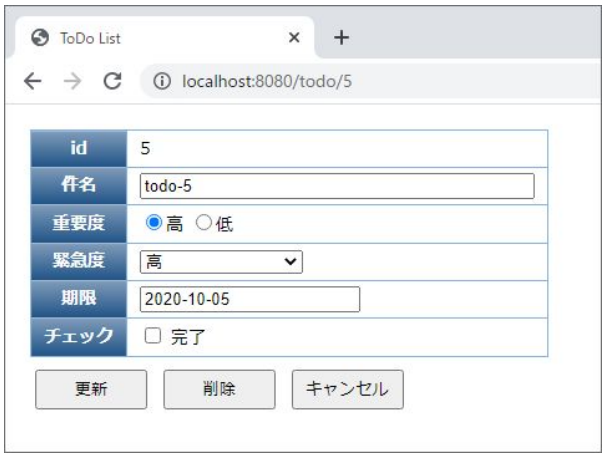
数の内容でレコードを更新します。ToDoは@Idをidに付与しているので、この内容で判断されます。

追加/更新をidで実行し分ける仕組みを説明します。まず次の２つの画面を見てください。

■[新規追加]で遷移してきたとき　⇒　id欄が空白



■件名リンクで遷移してきたとき　⇒　クリックしたToDoのidが表示されている



このように[新規追加]ではidが表示されず、件名クリックでは表示されることがわかります。

idに着目して処理の流れを見ていきます。まず登録の場合です。

1)入力画面には@GetMapping("/todo/create")のハンドラーメソッドから、初期状態のTodoDataオブジェクトが渡される(【リスト8-3】の※a)。

⇒idはnull。このためid欄は空白になる。

2)[登録]ボタンをクリックすると、@PostMapping("/todo/create")のハンドラーメソッドでTodoDataオブジェクトにバインドされる。

⇒入力画面ではidを変更できないのでnullのまま。

3)toEntity()でTodoオブジェクトを作成する。

⇒idはnullが引き継がれる。

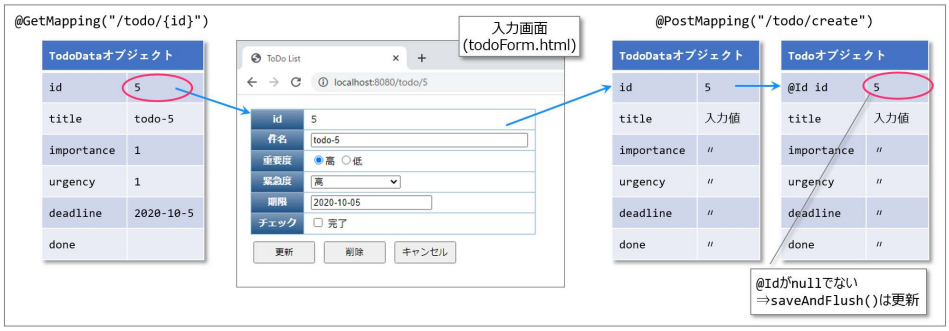
4)saveAndFlush()を実行する。

⇒@Idのプロパティ(=id)がnullのため、引数の内容がtodoテーブルに**追加**される。

⇒追加時、idの値が自動設定される(id列はSERIAL型)。

【図8-5】登録時、idはnullのまま

これに対し更新では、ハンドラーメソッドでToDo検索結果を入力画面に渡します(【リスト8-2】の※b)。idは選択されたToDoのものであり、下図のようにsaveAndFlush()の引数になるまで引き継がれます。よって**更新**処理となります。



【図8-6】更新時、idは非nullが引き継がれる

このとき実行されるUPDATE文は次のような形です。

```
UPDATE  todo  SET  title  =  todo.title, ...  WHERE  id  =  todo.id
```

> 入力画面には"todoData"という名前で、登録時TodoDataオブジェクト、更新時Todoオブジェクトを渡しています。オブジェクトの型が異なっていますが、Thymeleafはそのことを意識していません。どちらの場合も、渡されたオブジェクトから、*{プロパティ名}でプロパティ値を取得できるよう、同じ名前のプロパティを定義しておきます。

8.3 レコードを削除する

コントローラーの最後は、[削除]ボタンクリック時の処理です。これはdeleteById()を呼び出すシンプルなものです。

【リスト8-5】

com.example.todolist.controller.TodoListController.java#deleteTodo()

```
@PostMapping("/todo/delete")
public String deleteTodo(@ModelAttribute TodoData todoData) {
    todoRepository.deleteById(todoData.getId());
    return "redirect:/todo";
}
```

deleteById()も自動生成されるメソッドです(【表6-5】参照)。そろそろ名前から推測できるかもしれませんが、このメソッドは@Idを指定したプロパティを条件にしてレコードを削除します。これは次のようなDELETE文に相当します。

```
DELETE FROM todo WHERE id = todoData.getId()の値
```

8.4 Thymeleafでセッションのデータを参照する

入力画面は、前章のものに以下の変更を加えています。
【変更点1】表示項目にidを追加。ただし値はhidden要素に格納する。
【変更点2】modeによって表示するボタンを切り替える。

【リスト8-6】src/main/resources/templates/todoForm.html(一部抜粋)

```
<body>
  <form th:action="@{/}" method="post" th:object="${todoData}">
    <table>
      <!-- TodoList3で追加 開始 -->
      <tr>
        <th>id</th>
        <td>
          <span th:text="*{id}"></span>
          <input type="hidden" th:field="*{id}">     <!-- ① -->
        </td>
      </tr>
      <!-- TodoList3で追加 終了 -->
      <tr>
        <th>件名</th>
        <td>
          <input type="text" name="title" size="40" th:value="*{title}">
          <div th:if="${#fields.hasErrors('title')}" th:errors="*{title}"
               th:errorclass="red"></div>
        </td>
      </tr>
        :
    </table>
    <!-- TodoList3で変更 開始 -->
    <div th:if="${session.mode == 'update'}">      <!-- ② -->
      <button type="submit" th:formaction="@{/todo/update}">更新</button>
      <button type="submit" th:formaction="@{/todo/delete}">削除</button>
```

```
      <button type="submit" th:formaction="@{/todo/cancel}">キャンセル
</button>
    </div>
    <div th:unless="${session.mode == 'update'}">  <!-- ②③ -->
      <button type="submit" th:formaction="@{/todo/create}">登録</button>
      <button type="submit" th:formaction="@{/todo/cancel}">キャンセル
</button>
    </div>
    <!-- TodoList3で変更 終了 -->
  </form>
</body>
</html>
```

【変更点1】のidは、すでに説明したように登録/更新の判別、および削除時の条件に使います。

本来このidは、ユーザーに見せなくてよい項目ですが、挙動がわかるようspan要素としています。しかしspan要素はサーバーへ送信されないため、ブラウザに表示されないhidden要素を追加し、ここにidの値を格納します(①)。これでボタンクリック時、idもサーバーへ送られます(hidden要素については、【表4-2】参照)。

【変更点2】のmodeは、コントローラーでセッションに書き込んだ、表示するボタンを決めるものです(【表8-1】参照)。

セッションの値を変数式で参照するときは${session.名称}という形にします(②)。

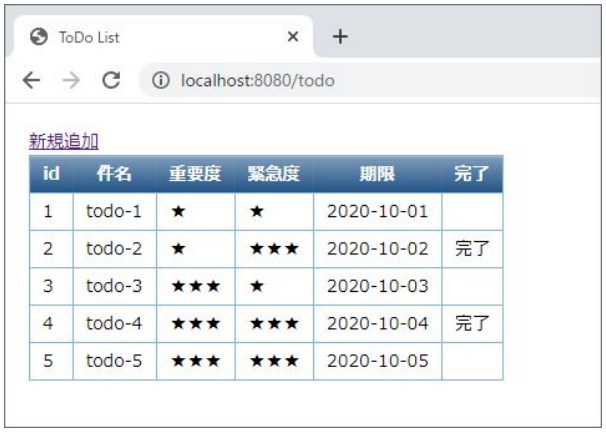
また**th:unless属性**(③)は、th:ifの反対で「条件式がfalseの場合」この要素を作成します。

> ThymeleafにはJavaのif～then～else文に相当する属性はありません。すでに説明した三項演算子(?:)は属性の中で使える演算子の一種です。
>
> その代わりswitch～case文に相当するth:switch, th:caseがあります。これで代替可能なケースも多いと思いますので、興味がある方は調べてみてください。

8.5 PRG(Post-Redirect-Get)パターン

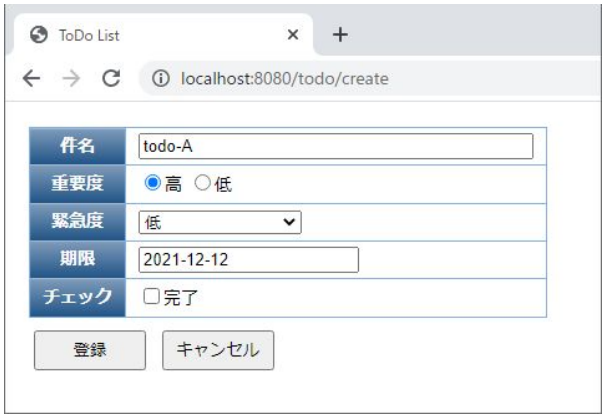
前章の登録用ハンドラーメソッドcreateTodo()は、ToDo登録後、showTodoList()を呼び出して一覧画面を再表示しました。しかし、この方法には問題があります。これは以下の手順で再現できます。

1)前章で作成したTodolist2を起動し、http://localhost:8080/todoをアクセスする。

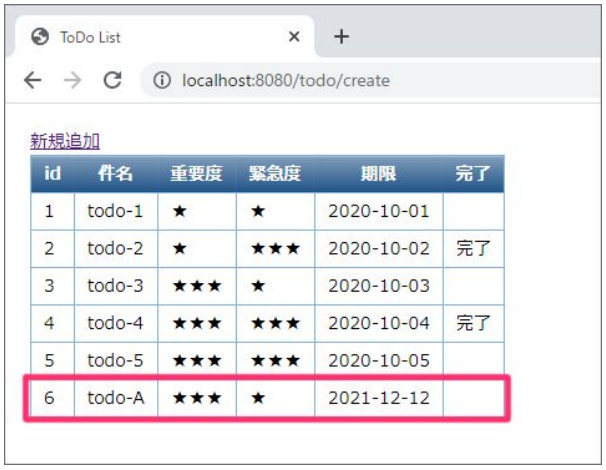


2)[新規追加]をクリックする

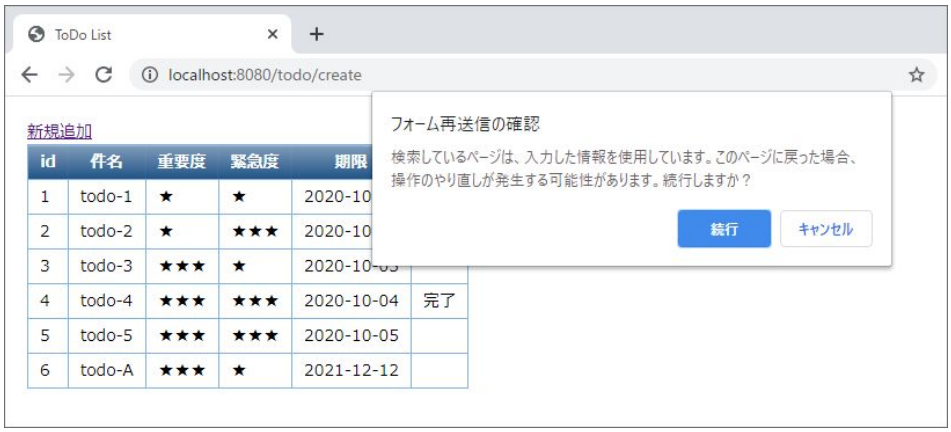
3)データを入力して[登録]ボタンをクリックする。



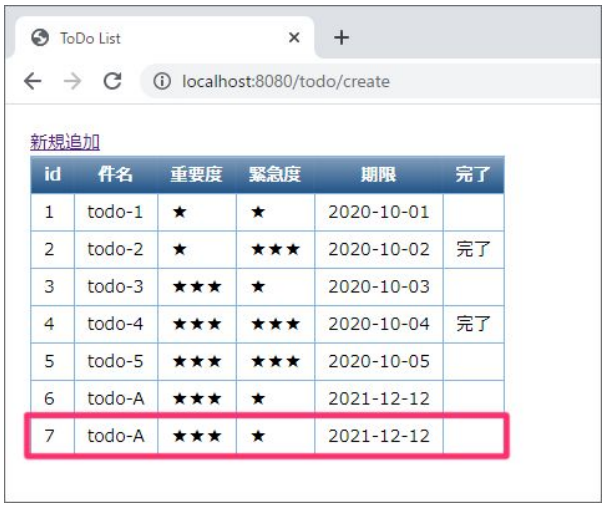
4)一覧画面に登録した内容が追加されている。



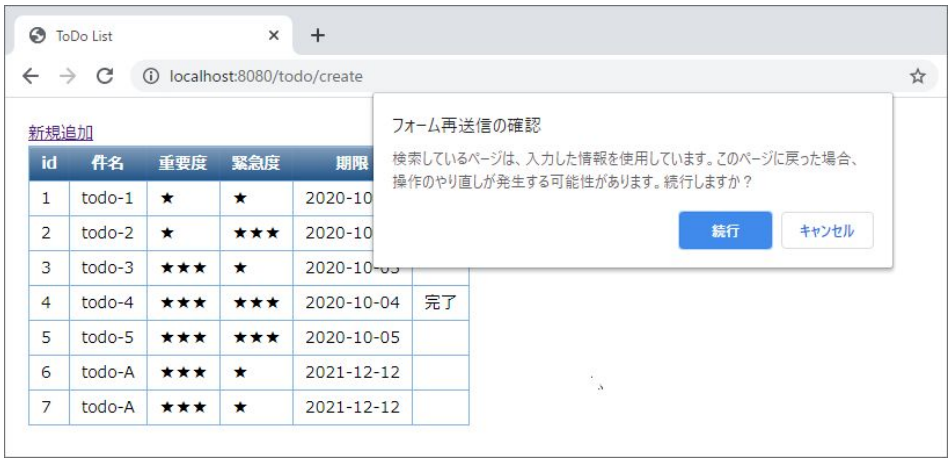
5)ブラウザの再読み込みボタンをクリック、または[CTRL]+[R]を押下する。
⇒「フォーム再送信の確認」ダイアログが表示されるので、[続行]ボタンをクリックする。



6)3)で入力したものが、もう1件追加される。

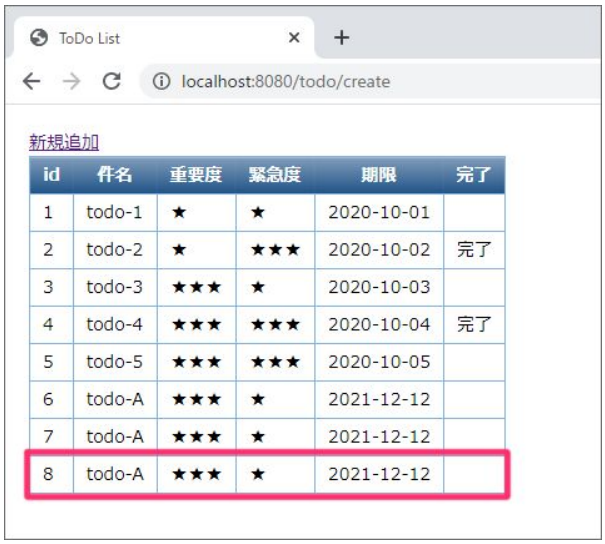


7)ブラウザの再読み込みボタンをクリック、または[CTRL]+[R]を押下する。
⇒「フォーム再送信の確認」ダイアログが表示されるので、[続行]ボタンをクリックする。



id	件名	重要度	緊急度	期限	
1	todo-1	★	★	2020-10	
2	todo-2	★	★★★	2020-10	
3	todo-3	★★★	★	2020-10-03	
4	todo-4	★★★	★★★	2020-10-04	完了
5	todo-5	★★★	★★★	2020-10-05	
6	todo-A	★★★	★	2021-12-12	
7	todo-A	★★★	★	2021-12-12	

8)3)で入力したものが、さらにもう1件追加される。



id	件名	重要度	緊急度	期限	完了
1	todo-1	★	★	2020-10-01	
2	todo-2	★	★★★	2020-10-02	完了
3	todo-3	★★★	★	2020-10-03	
4	todo-4	★★★	★★★	2020-10-04	完了
5	todo-5	★★★	★★★	2020-10-05	
6	todo-A	★★★	★	2021-12-12	
7	todo-A	★★★	★	2021-12-12	
8	todo-A	★★★	★	2021-12-12	

：

このように登録直後に再読み込みをすると、入力画面を通らず同じデータが登録されてしまいます。これは明らかに不具合でしょう。

なぜこういうことが起こるのか？まずTodolist2のコードを再掲します。

【リスト8-7】Todolist2のTodoListController.java#createTodo()

```
  @PostMapping("/todo/create")
```

```
  public ModelAndView createTodo(@ModelAttribute @Validated TodoData
todoData,
                                  BindingResult result, ModelAndView mv) {
      // エラーチェック
      boolean isValid = todoService.isValid(todoData, result);
      if (!result.hasErrors() && isValid) {
        // エラーなし
        Todo todo = todoData.toEntity();
        todoRepository.saveAndFlush(todo);
        return showTodoList(mv);                //  ①

      } else {
          // エラーあり
          mv.setViewName("todoForm");
          // mv.addObject("todoData", todoData);
          return mv;                             //  ②
      }
  }
```

ToDo登録後、①のshowTodoList()で一覧を再表示させますが、表示されたブラウザのアドレス欄は/todo/crate/のままです。これは先ほどの操作3)→4)を見るとわかります。この状態で再読み込みすると、ブラウザは直前のリクエスト(=アドレス欄のリクエスト)を再度実行しようとします。5),7)の「フォーム再送信の確認」ダイアログは、その可否を確認するものです。[続行]とすると、直前の/todo/crate/がPOSTリクエストでサーバーへ送られてしまい、余計なToDoが追加されるのです。

解決方法はいくつか考えられますが、ここでは①を「/todoへのリダイレクト」へ変更します。リダイレクトを実行すると、アドレス欄はリダイレクト先(/todo)に変わります。そして、リダイレクト先のハンドラーメソッドshowTodoList()で一覧を再表示します。

```
    return "redirect:/todo";
```

しかしこれだけではModelAndView型オブジェクトを返す②と共存できません。そこで②もStringで次画面を指定するよう変更します。

```
    return "todoForm";
```

するとModelAndViewの持つ「ビュー名を保持する」という機能は不要になります。こういった場合、「ビューに渡すデータを保持する」だけのModel(org.springframework.ui.Model)を使うことができます。これで書き換えたのが、下記のcreateTodo()です。ハンドラーメソッドの戻り値もString型に変更しています。

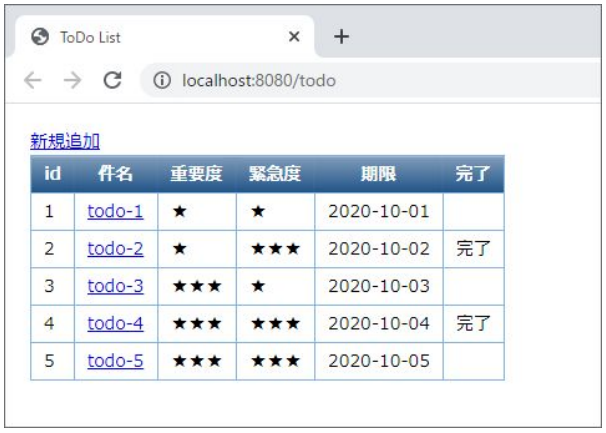
【リスト8-8】Todolist3のTodoListController.java#createTodo()

```
    // ToDo追加処理(Todolist2で追加したものをTodolist3で改善)
    @PostMapping("/todo/create")
    public String createTodo(@ModelAttribute @Validated TodoData todoData,
                             BindingResult result,
                             Model model) {
        // エラーチェック
        boolean isValid = todoService.isValid(todoData, result);
        if (!result.hasErrors() && isValid) {
            // エラーなし
            Todo todo = todoData.toEntity();
            todoRepository.saveAndFlush(todo);
            return "redirect:/todo";

        } else {
            // エラーあり
            // model.addAttribute("todoData", todoData);
            return "todoForm";
        }
}
```

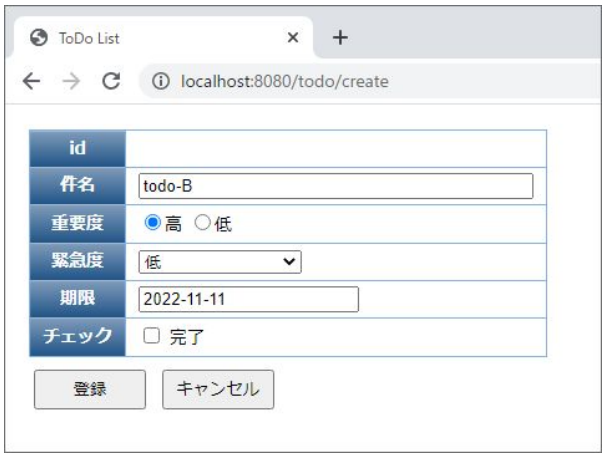
今度は本章のTodolist3で、登録直後の再読み込みを実行してみます。

1)Todolist3を起動し、http://localhost:8080/todoをアクセスする。

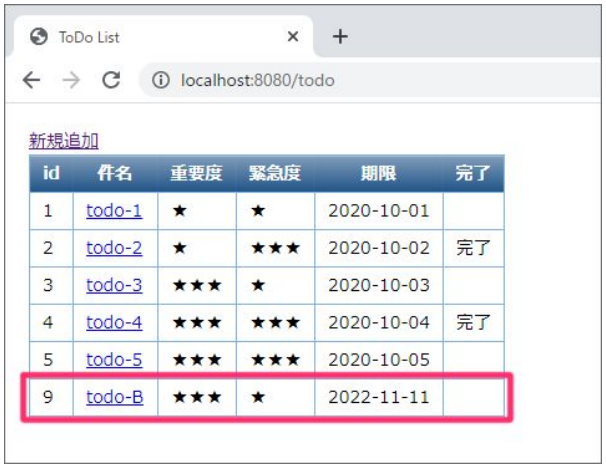
2)[新規追加]をクリックする



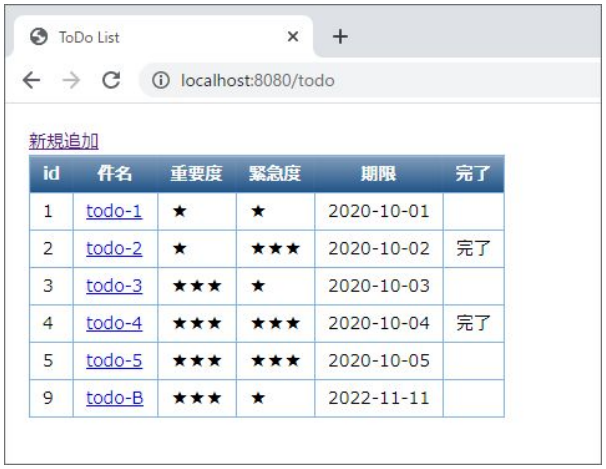
4)データを入力して[登録]ボタンをクリックする。



5)一覧画面に登録した内容が追加されている。

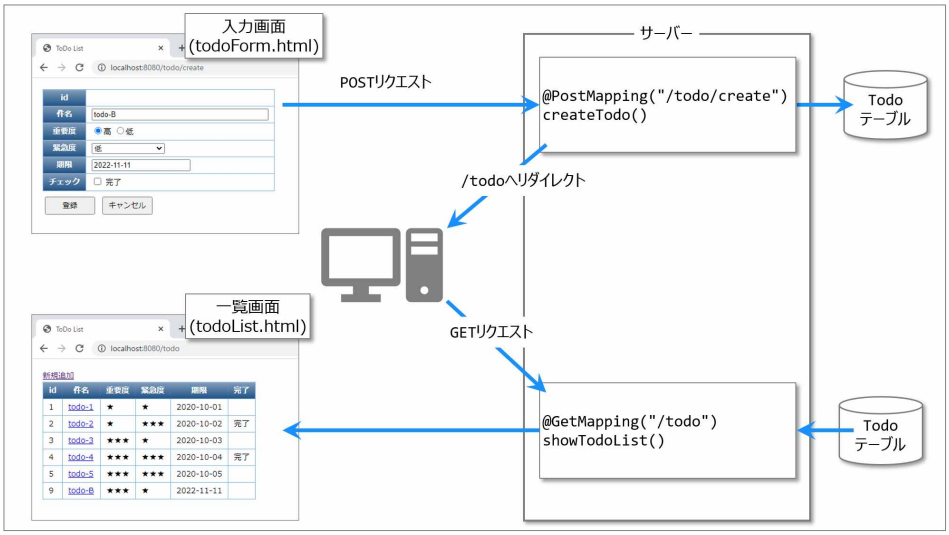


6)ブラウザの再読み込みボタンをクリック、または[CTRL]+[R]を押下する。
⇒一覧画面が再表示される。



今度は「フォーム再送信の確認」ダイアログが表示されません。何回再読み込みしても、一覧が表示されるだけです。意図しないデータ追加を回避することができました。

このように登録処理後、リダイレクトで次画面へ遷移させる方法を**「PRG(Post-Redirect-Get)パターン」**と呼んでいます。図で表すと次のようなイメージです。



【図8-7】登録処理のPRGパターン動作イメージ

ポイントは、リダイレクトを実行するとクライアントPC(ブラウザ)へ一度制御が戻る点です。ここからリダイレクト先へGETリクエストが自動的に送られます。操作していると1往復のように見えますが、実際には2往復しています。こうすれば再読み込みしても、直前

のリクエストは一覧画面を表示するGETリクエストになるので、これが再実行されるわけです。

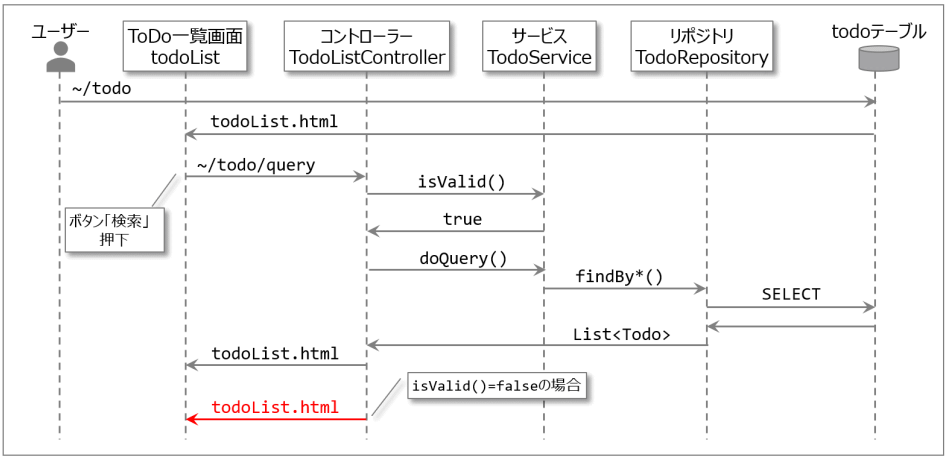
> このPRGパターンは、同じフォームデータが複数回サーバーへ送信されないようにする方法の１つです。しかし、多重登録の原因となる操作パターンは他にもあるため、さらに別な防止策も必要です。興味がある方は「二重サブミット」などをキーワードに調べてみてください。

　これでToDoの登録、変更、削除ができるようになりました。ただ、表示は一覧だけです。件数が多くなってくると、不便です。そこで次章からは、この検索機能を強化していきます。

9．入力された条件で検索する

本章では条件に一致するToDoを検索できるようにします。条件入力画面を追加する方法もありますが、ここでは一覧画面(todoList.html)に検索条件フォーム(form要素)を追加します。

【図9-1】ToDoの検索操作



※~/todoの詳細は6章参照(【図6-3】)

【図9-2】検索操作のシーケンス

作成するプロジェクトの仕様

プロジェクト名	Todolist4
依存関係	Spring Web, Spring Boot DevTools, Thymeleaf, Lombok, Spring Data JPA, PostgreSQL Driver, Validation

```
Todolist4
├ src/main/java
│  ├ com.example.todolist
│  │  └ TodolistApplication.java
│  ├ com.example.todolist.common(★)
│  │  └ Utils.java(★)
│  ├ com.example.todolist.controller
│  │  └ TodoListController.java(▲)
│  ├ com.example.todolist.entity
│  │  └ Todo.java
│  ├ com.example.todolist.form
│  │  ├  TodoData.java
│  │  └  TodoQuery.java(★)
│  ├ com.example.todolist.repository
│  │  └ TodoRepository.java(▲)
│  └ com.example.todolist.service
│      └ TodoService.java(▲)
└ src/main/resources
   ├ static
   │  └ css
   │     └ style.css
   └ templates
       ├ todoForm.html(▲)
       └ todoList.html(▲)
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

9.1 検索条件フォームの追加

一覧画面に、検索条件を入力するためのフォーム(form要素)を追加します。

【リスト9-1】src/main/resources/templates/totoList.html(一部抜粋)

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>ToDo List</title>
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css">
</head>
<body>
  <!-- Todolist4で追加 開始 -->
  <form th:action="@{/}" method="post" th:object="${todoQuery}"><!-- ① -
->
    <div style="display: flex">
      <table border="1">
        <tr>
          <th>件名</th>
          <th>重要度</th>
          <th>緊急度</th>
          <th>期限</th>
          <th>完了</th>
        </tr>
        <tr>
          <td>
            <input type="text" name="title" size="40" th:value="*
{title}">
          </td>
          <td>
            <select name="importance">
              <option value="-1" th:field="*{importance}">-</option>
              <option value="1" th:field="*{importance}">高</option>
```

```
            <option value="0" th:field="*{importance}">低</option>
          </select>
        </td>
        <td>
          <select name="urgency">
            <option value="-1" th:field="*{urgency}">-</option>
            <option value="1" th:field="*{urgency}">高</option>
            <option value="0" th:field="*{urgency}">低</option>
          </select>
        </td>
        <td>
          <input type="text" name="deadlineFrom" th:value="*
{deadlineFrom}" size="10"
                 placeholder="yyyy-mm-dd">
          ～
          <input type="text" name="deadlineTo" th:value="*{deadlineTo}"
size="10"
                 placeholder="yyyy-mm-dd">
        </td>
        <td>
          <input type="checkbox" value="Y" th:field="*{done}">完了
        </td>
      </table>
      <button type="submit" th:formaction="@{/todo/query}">検索</button>
    </div>
    <div th:if="${#fields.hasErrors('deadlineFrom')}" th:errors="*
{deadlineFrom}"
        th:errorclass="red"></div>
    <div th:if="${#fields.hasErrors('deadlineTo')}" th:errors="*
{deadlineTo}"
        th:errorclass="red"></div>
    <hr>
    <button type="submit" th:formaction="@{/todo/create/form}">新規追加
</button>
```

```
  </form>
  <!-- Todolist4で追加 終了 -->
  <table border="1">
    <tr>
      <th>id</th>
      <th>件名</th>
      :
</body>
</html>
```

①のth:object="${todoQuery}"からわかるように、このフォームとコントローラーはtodoQueryという名前のオブジェクトで関連付けます(定義は後述します)。

デザイン的には入力画面の項目を横並びにしたような形です。違いは期限を範囲で表せるように、開始と終了の2つ用意していることです(以下、期限：開始、期限：終了と呼びます)。

このフォームに入力された内容から、下表のような検索を行います。

【表9-1】検索条件と処理内容

優先度	検索条件	処理内容
1	件名に文字(列)が入力されている	その文字(列)を件名に含むToDoを検索する
2	緊急度="高", "低"	該当する緊急度のToDoを検索する
3	重要度="高", "低"	該当する重要度のToDoを検索する
4	期限：開始!=""、期限：終了=""	期限：開始 ≦ 期限 に該当するToDoを検索する
5	期限：開始=""、期限：終了!=""	期限 ≦ 期限：終了 に該当するToDoを検索する
6	期限：開始!=""、期限：終了!=""	期限：開始 ≦ 期限 ≦ 期限：終了 に該当する ToDo検索する
7	完了がチェックされている	完了='Y' のToDo検索する
8	上記以外	全ToDoを検索する

入力内容は、優先度1 →2 →...→8の順にチェックし、該当すればその条件で検索します。このため「緊急度="高"　かつ　重要度="高"のToDo」といった複合条件による検索はできません(緊急度="高"のものを検索)。このような検索は次章で対応します。

また一覧画面では、新規追加をリンクからボタンに変更しています。

```
<a href="/todo/create">新規追加</a>
      ↓
<button type="submit" th:formaction="@{/todo/create/form}">新規追加
</button>
```

これに合わせて入力画面(todoForm.html)の[登録]ボタンも、以下のように変更しています。

【リスト9-2】src/main/resources/templates/todoForm.html(変更箇所)

```
<button type="submit" th:formaction="@{/todo/create}">登録</button>
      ↓
<button type="submit" th:formaction="@{/todo/create/do}">登録</button>
```

これでURLパスと対応する処理は下表のようになります。

【表9-2】入力に関するURLパス

URLパス	処理内容
/todo/create/form	入力画面を表示する
/todo/create/do	入力内容を登録する

> これは検索処理とは関係ないデザイン上の変更です。

9.2 検索条件の取得

次に検索条件フォームへ入力された内容をバインドするフォームクラスを追加します。

【リスト9-3】com.example.todolist.form.TodoQuery.java

```
package com.example.todolist.form;

import lombok.Data;

@Data
public class TodoQuery {
    private String title;
    private Integer importance;
    private Integer urgency;
    private String deadlineFrom;
    private String deadlineTo;
    private String done;

    public TodoQuery() {
        title = "";
        importance = -1;
        urgency = -1;
        deadlineFrom = "";
        deadlineTo = "";
        done = "";
    }
}
```

ToDo入力用のTodoDataと違い、検索条件に対しては必須入力といったチェックをしないので、バリデーション用のアノテーション(@NotBlank,@NotNull,@Min)はありません。ただし期限：開始、期限：終了については、TodoServiceでチェックをします(後述)。

コントローラーの検索処理は、以下のようになっています。

【リスト9-4】com.example.todolist.controller.TodoListController.java関連箇所

```
    @GettMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv) {
        // 一覧を検索して表示する
        mv.setViewName("todoList");
        List<Todo> todoList = todoRepository.findAll();
        mv.addObject("todoList", todoList);
        mv.addObject("todoQuery", new TodoQuery());         //
※Todolist4で追加
        return mv;
    }
      :
    @PostMapping("/todo/query")
    public ModelAndView queryTodo(@ModelAttribute TodoQuery todoQuery,
//  ①
                                  BindingResult result,
//  ②
                                  ModelAndView mv) {
        mv.setViewName("todoList");

        List<Todo> todoList = null;
        if (todoService.isValid(todoQuery, result)) {   //  ③
            // エラーが無ければ検索
            todoList = todoService.doQuery(todoQuery);   //  ④
        }
        // mv.addObject("todoQuery", todoQuery);         //  ⑤
        mv.addObject("todoList", todoList);             //  ⑥

        return mv;
    }
```

showTodoList()

最初に表示する一覧画面の処理です。

todoListには初期化したTodoQueryオブジェクトを渡します(※部分)。これで検索条件フォームは初期状態で表示されます。

queryTodo()

[検索]ボタンが押された時の処理です。

検索条件フォームの内容はtodoQueryオブジェクトにバインドされます(①)。

TodoQueryにはバリデーションが無いので@Validatedを指定しません。しかしTodoServiceでのチェック結果を格納するためBindingResultは引数に残しておきます(②)。

独自チェックはTodoService#isValid()で行います(③)。エラーが無ければ入力された条件で検索します(④)。詳細は次節で説明します。

エラーがあった場合、todoListはnullのままなので、一覧には何も表示されません。

また⑤のaddObject()は、前章で説明したように以下の理由で省略可能です。

- todoQueryが@ModelAttributeでバインドしたものである
- 第1引数の名前がクラス名の先頭を小文字にしたものである
- 遷移先todoList.htmlのth:objectでもこの名前("todoQuery")を指定している

以下は期限：開始、期限：終了に対するチェック処理です。なおTodoServiceには、すでに6章のTodolist2でisValid()を作成しています。ただし引数の型・数が違うので、同名メソッドを追加できます(オーバーロード)。

【リスト9-5】com.example.todolist.service.TodoService.java#isValid()

```
package com.example.todolist.service;
      :
public class TodoService {
      :
    // Todolist4で追加
    public boolean isValid(TodoQuery todoQuery, BindingResult result) {
        boolean ans = true;

        // 期限:開始の形式をチェック
        String date = todoQuery.getDeadlineFrom();
```

```
        if (!date.equals("")) {
            try {
                LocalDate.parse(date);
            } catch (DateTimeException e) {
                // parseできない場合
                FieldError fieldError = new FieldError(
                    result.getObjectName(),
                    "deadlineFrom",
                    "期限：開始を入力するときはyyyy-mm-dd形式で入力してくださ
い");
                result.addError(fieldError);
                ans = false;
            }
        }
        // 期限:終了の形式をチェック
        date = todoQuery.getDeadlineTo();
        if (!date.equals("")) {
            try {
                LocalDate.parse(date);
            } catch (DateTimeException e) {
              // parseできない場合
              FieldError fieldError = new FieldError(
                  result.getObjectName(),
                  "deadlineTo",
                  "期限：終了を入力するときはyyyy-mm-dd形式で入力してくださ
い");
              result.addError(fieldError);
              ans = false;
            }
        }
        return ans;
    }
      :
}
```

処理の流れはTodolist2で追加したisValid()と同じです。todoQueryの期限：開始、期限：終了にエラーがあればFieldErrorオブジェクトを生成し、引数のBindingResultオブジェクト(result)に追加します。

> 7.3節でも触れたようバリデーションを行うバリデーター(Validator)は独自に作成できます。一度作ってしまえば使い回せます。興味がある方は以下のインターフェース名をキーに調べてみてください。
>
> jakarta.validation.ConstraintValidator

9.3 検索処理の定義・実行

残っているのは入力に応じた検索を行う、TodoService#doQuery()の仕組みです。
本章では「入力された文字列を件名に含むもの」といった検索を行います(【表9-1】)。しかし該当するメソッドは【表6-5】にありません。それでは自分で書くのか？というと、それも違います。「ある命名規則に準じた抽象メソッドをリポジトリに宣言すると、Spring Bootが自動実装してくれる」という機能を利用します。

以下がこの章で使用するTodoRepositoryです。検索処理に対応する抽象メソッドを追加しています。

【リスト9-6】com.example.todolist.repository.TodoRepository.java

```
package com.example.todolist.repository;

import java.sql.Date;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.todolist.entity.Todo;

@Repository
public interface TodoRepository extends JpaRepository<Todo, Integer> {
    List<Todo> findByTitleLike(String title);
    List<Todo> findByImportance(Integer importance);
    List<Todo> findByUrgency(Integer urgency);
    List<Todo> findByDeadlineBetweenOrderByDeadlineAsc(Date from, Date
to);
    List<Todo> findByDeadlineGreaterThanEqualOrderByDeadlineAsc(Date
from);
    List<Todo> findByDeadlineLessThanEqualOrderByDeadlineAsc(Date to);
    List<Todo> findByDone(String done);
}
```

追加した抽象メソッドは【表9-1】の検索処理と１対１で対応しています。

【表9-3】検索処理とメソッド宣言

#	検索処理	メソッド宣言(戻り値はすべてList<Todo>)
1)	その文字(列)を件名に含む	**findBy**Title**Like**(String title)
2)	緊急度が一致する	**findBy**Importance(Integer importance)
3)	重要度が一致する	**findBy**Urgency(Integer urgency)
4)	期限が期限：開始～期限：終了の範囲内	**findBy**Deadline**BetweenOrderBy**Deadline**Asc**(Date from, Date to)
5)	期限：開始 ≦ 期限	**findBy**Deadline**GreaterThanEqualOrderBy**Deadline**Asc**(Date from)
6)	期限 ≦ 期限：終了	**findBy**Deadline**LessThanEqualOrderBy**Deadline**Asc**(Date to)
7)	完了が一致する	**findBy**Done(String done)

この命名規則を大雑把に言えば、以下のようになっています。

・"findBy～"は「～で検索する」を表す

・"OrderBy～"は「～で並べ替える」を表す

・～の部分は、Todoクラスのプロパティ(=todoテーブルの列)を表す

⇒findByのときは引数で条件値を渡す

上表のメソッド宣言のプロパティ名を～にすると次のようになります。

・findBy～(条件値)

・findBy～Like(条件値)

・findBy～BetweenOrderBy～Asc(条件値1, 条件値2)

・findBy～GreaterThanEqualOrderBy～Asc(条件値)

・findBy～LessThanEqualOrderBy～Asc(条件値)

> 本章で使用している命名規則はごく一部です。詳細については、以下のサイトなどを参照してください。
>
> Spring Data JPA - Reference Documentation > 5. JPA Repositories > 5.3.2. Query Creation
>
> https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#jpa.query-methods.query-creation

findBy～(条件値)

最もシンプルな2)のfindByImportance(Integer importance)から説明します。

findByに続くImportanceは、Todoクラスのimportanceプロパティのことです(キャメルケースのため大文字で始まる)。引数は探す値(=検索条件フォームの重要度に入力された値)です。これで「importanceが引数に一致するtodoレコードを検索する」SELECT文が実行されます。

```
findByImportance(Integer importance)
↓
SELECT * FROM todo WHERE importance = 引数importanceの値(検索条件フォーム入力値)
```

検索結果はList<Todo>になります。3), 7)も同様です。

> 【表6-5】のfindAll(Example<S> example)を使えば、2, 3), 7)と同等の検索ができます。
> たとえば以下のようにすると、2)のfindByImportance()と同じ結果が得られます。
>
> ```
> Todo todo = new Todo();
> todo.setImportance(todoQuery.getImportance());
> Example<Todo> example = Example.of(todo);
> todoList = todoRepository.findAll(example);
> ```
>
> 少し冗長な感じがするので本書では使っていません。興味がある方は調べてみてください。

findBy～Like(条件値)

1)のfindByTitleLikeは「titleが指定された文字(列)を含むもの」という意味です。SQLで言えばLike演算子による検索となります。

```
findByTitleLike(String title)
↓
SELECT * FROM todo WHERE title Like '引数titleの値'
```

引数には必要に応じてワイルドカード(%)を追加します。

【表9-4】ワイルドカードの使い方

引数	意味(検索条件)
"xyz%"	"xyz"で始まるもの(前方一致)
"%xyz"	"xyz"で終わりもの(後方一致)
"%xyz%"	"xyz"を含むもの(部分一致)
"xyz"	"xyz"であるもの(完全一致)

> likeのほかにもワイルドカード(%)を使わないStartingWith(前方一致), EndingWith(後方一致), Containing(部分一致)などもあります。興味がある方は調べてみてください。

findBy～Between, OrderBy～Asc

4)のfindByDeadlineBetweenOrderByDeadlineAscはfindBy～BetweenとOrderBy～Ascの組み合わせです。

まずfindByDeadlineBetweenは「deadlineが引数の範囲内に含まれるもの」を表します。これもSQLで言えばBetween演算子による検索です。

```
findByDeadlineBetween(Date from, Date to)
↓
SELECT * FROM todo WHERE deadline BETWEEN 引数fromの値 AND 引数toの値
```

OrderByDeadlineは、検索結果をdeadlineで並べ替えます。最後のAscは、その並べ方の指定です。

Asc ：～の昇順にする(値が小さい→大きい順；ascendingの略)
Desc：～の降順にする(値が大きい→小さい順；descendingの略)

これで**findBy**Deadline**BetweenOrderBy**Deadline**Asc**()は「deadlineが引数の範囲内に含まれるレコードを検索し、それをdeadlineの昇順に並べ替えて返す」と読み解くことができます(ここは期限が小さい、つまり期限が早いToDoから表示させるのが自然だと思います)。SELECT文で表すと、以下のようになります。

```
findByDeadlineBetweenOrderByDeadlineAsc(Date from, Date to)
↓
SELECT * FROM todo WHERE deadline BETWEEN 引数fromの値 AND 引数toの値
```

```
                    ORDER  BY  deadline  ASC
```

> PostgreSQLをはじめとするリレーショナルデータベース(RDB)では、ORDER BYを指定しない限り、検索結果の並び順は不定、と考えた方が無難です。並びが主キーの昇順に見える場合もありますが、それは「たまたま」そうなっているだけです。レコードを更新すると崩れます。
> 検索処理を実行するときは、結果の並び順も意識しましょう。

findBy〜GreaterThanEqual, findBy〜LessThanEqual

5), 6)のGreaterThanEqual, LessThanEqualは、それぞれ「以上」、「以下」を表します。

```
findByDeadlineGreaterThanEqualOrderByDeadlineAsc(Date from)
↓
SELECT  *  FROM  todo  WHERE  deadline  >=  引数fromの値
                  ORDER  BY  deadline ASC
```

```
findByDeadlineLessThanEqualOrderByDeadlineAsc(Date to)
↓
SELECT  *  FROM  todo  WHERE  deadline  <=  引数toの値
                  ORDER  BY  deadline  ASC
```

> 本書では使用しませんが、条件をAndやOrで複数記述することもできます。
> findByImportanceAndUrgency(Integer importance, Integer urgency)
> ↓
> SELECT * FROM todo WHERE importance = 引数importanceの値
> AND urgency = 引数urgencyの値
> 詳細は上記Reference Documentationなどを参照してください。

検索処理は、入力された条件から対応するこれらのメソッドを呼び出すことで実現します。これも分岐が多い処理なのでコントローラーではなくサービスに記述します。

【リスト9-7】com.example.todolist.service.TodoService.java#doQuery()

```
package com.example.todolist.service;
    :
```

```
@Service
@AllArgsConstructor      //  ①
public class TodoService {
      :
  private final TodoRepository todoRepository;    //  ①
      :
  // Todolist4で追加
  public List<Todo> doQuery(TodoQuery todoQuery) {
    List<Todo> todoList = null;
    if (todoQuery.getTitle().length() > 0) {
      // タイトルで検索
      todoList = todoRepository.findByTitleLike("%" +
todoQuery.getTitle() + "%");

    } else if (todoQuery.getImportance() != null &&
todoQuery.getImportance() != -1) {
      // 重要度で検索
      todoList =
todoRepository.findByImportance(todoQuery.getImportance());

    } else if (todoQuery.getUrgency() != null && todoQuery.getUrgency() != -1) {
      // 緊急度で検索
      todoList = todoRepository.findByUrgency(todoQuery.getUrgency());

    } else if (!todoQuery.getDeadlineFrom().equals("") &&
              todoQuery.getDeadlineTo().equals("")) {
      // 期限 開始～
      todoList = todoRepository
        .findByDeadlineGreaterThanEqualOrderByDeadlineAsc(
          Utils.str2date(todoQuery.getDeadlineFrom()));

    } else if (todoQuery.getDeadlineFrom().equals("") &&
              !todoQuery.getDeadlineTo().equals("")) {
      // 期限 ～終了
```

```
      todoList = todoRepository
        .findByDeadlineLessThanEqualOrderByDeadlineAsc(
            Utils.str2date(todoQuery.getDeadlineTo()));

    } else if (!todoQuery.getDeadlineFrom().equals("") &&
              !todoQuery.getDeadlineTo().equals("")) {
      // 期限 開始～終了
      todoList = todoRepository
        .findByDeadlineBetweenOrderByDeadlineAsc(
             Utils.str2date(todoQuery.getDeadlineFrom()),
             Utils.str2date(todoQuery.getDeadlineTo()));

    } else if (todoQuery.getDone() != null &&
todoQuery.getDone().equals("Y")) {
      // 完了で検索
      todoList = todoRepository.findByDone("Y");

    } else {
      // 入力条件が無ければ全件検索
      todoList = todoRepository.findAll();
    }

    return todoList;
  }
    :
}
```

このメソッドはリポジトリを使うので、コンストラクターインジェクションで取得します(①)。

またUtils#str2date()は、文字列をjava.sql.Date型に変換するヘルパーメソッドです。次章以降、他クラスでも必要になるので独立させておきます

【リスト9-8】com.example.todolist.common.Utils.java

```
package com.example.todolist.common;
```

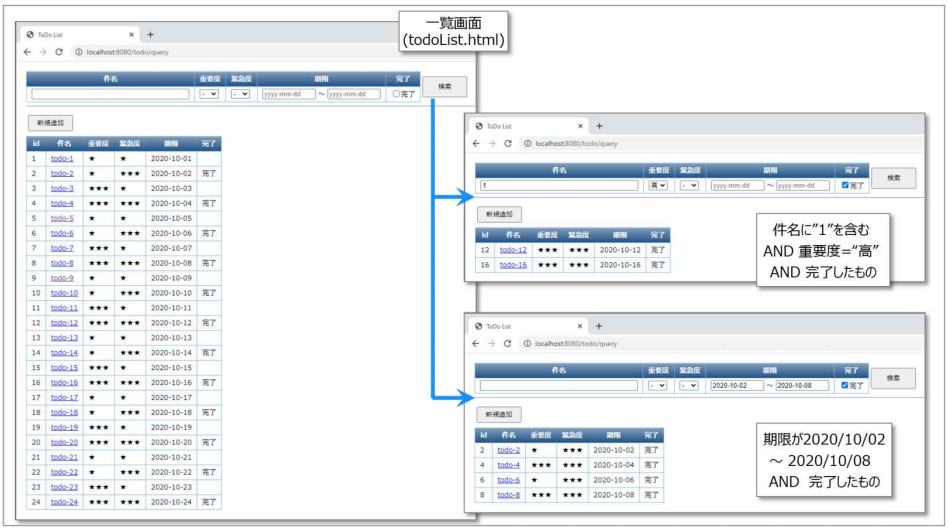
```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.sql.Date;

public class Utils {
    // Todolist4で追加
    private static final SimpleDateFormat sdf = new
SimpleDateFormat("yyyy-MM-dd");

    public static  Date str2date(String s) {
        long ms = 0;
        try {
            ms = sdf.parse(s).getTime();
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return new Date(ms);
    }
}
```

フォームに入力された条件で、いろいろな検索ができるようになりました。しかし「緊急度="高"　かつ　重要度="高"　かつ　未完了のToDo」といった検索はできません。これでは期限に遅れてクレームになるかもしれません。反対に「緊急度="低"　かつ　重要度="低"　かつ　未完了のToDo」は、本当にやるべきか考える余地がありそうです。

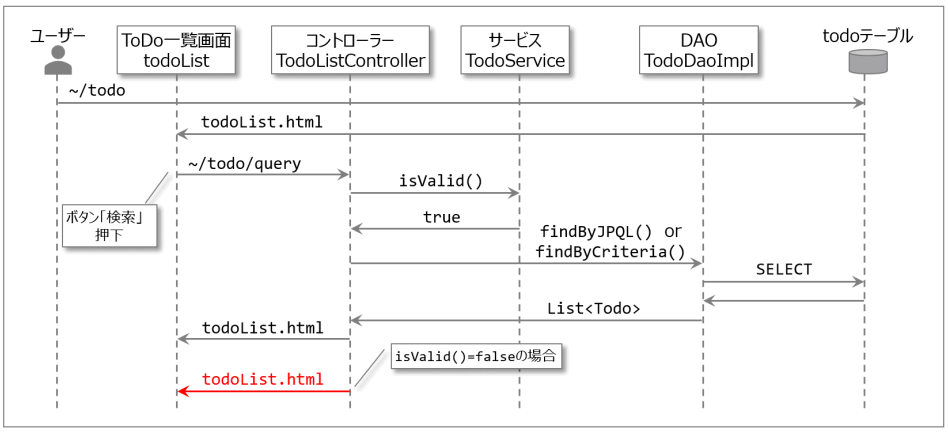
このように検索パターンは様々ですが、それを本章の内容だけ実現するのは大変です。次章ではもっと柔軟な検索処理を導入します。

10．動的なクエリによる検索

前章の検索処理は、条件を複数入力しても１つしか使いませんでした。これを本章では、条件が2つなら「条件1　AND　条件2」、3つなら「条件1　AND 条件2　AND 条件3」といった検索ができるようにします。つまり実行時に、入力内容から条件を組み立てて検索します。こういった処理を「動的なクエリ(問い合わせ)」と言います。



【図10-1】ToDoの動的な検索操作



※~/todoの詳細は6章参照(【図6-3】)

【図10-2】動的な検索操作のシーケンス

作成するプロジェクトの仕様

プロジェクト名	Todolist5
依存関係	Spring Web, Spring Boot DevTools, Thymeleaf, Lombok, Spring Data JPA, PostgreSQL Driver, Validation

```
Todolist5
├ src/main/java
│  ├ com.example.todolist
│  │  └ TodolistApplication.java
│  ├ com.example.todolist.common
│  │  └ Utils.java
│  ├ com.example.todolist.controller
│  │  └ TodoListController.java(▲)
│  ├ com.example.todolist.dao(★)
│  │  ├ TodoDao.java(★)
│  │  └ TodoDaoImpl.java(★)
│  ├ com.example.todolist.entity
│  │  ├ Todo.java
│  │  └ Todo_.java(★自動生成)
│  ├ com.example.todolist.form
│  │  ├  TodoData.java
│  │  └  TodoQuery.java
│  ├ com.example.todolist.repository
│  │  └ TodoRepository.java
│  └ com.example.todolist.service
│      └ TodoService.java
└ src/main/resources
   ├ static
   │  └ css
   │    └ style.css
   └ templates
      ├ todoForm.html
      └ todoList.html
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

10.1 動的なクエリ実行方法

前章で説明したように、リポジトリに抽象メソッドを宣言すると、そのメソッド名を手掛かりにして検索処理が自動実装されます。検索条件をAnd/Orで結合すれば、ある程度複雑なパターンにも対応できます。しかしそれにも限度があります。

たとえばToDoの検索パターンは、単純に考えても144通り存在します。

【表10-1】ToDoの検索パターン

項目	入力パターン			パターン数
件名	入力・なし	入力・あり		2
重要度	指定・なし	低	高	3
緊急度	指定・なし	低	高	3
期限：開始	入力・なし	入力・あり		2
期限：開始	入力・なし	入力・あり		2
完了	未チェック	チェック		2

検索パターン　＝　2　×　3　×　3　×　2　×　2　×　2　＝　144通り

全パターンをfindBy～And～の形でリポジトリに宣言し、それらを条件分岐で実行し分けるのは(プログラム的には可能ですが)現実的な方法とは言えないでしょう。

こういう場合Spring Bootでは、実行時に検索条件を作成する「動的なクエリ」という機能が利用できます。実行方法は複数ありますが、本書では次の2つを取り上げます。

JPQL(Java Persistence Query Language)

Spring Boot内部で使用している問い合わせ言語(クエリ言語)です。SELECT文とよく似ていますが、文法上の操作対象はエンティティオブジェクトです。エンティティはテーブルに関連付けられているので、最終的にテーブルの検索となります。

このJPQLは文字列として作成します。

> JPQLを@NamedQueryアノテーションで実行することもできます。しかし、実行内容を動的に変えられないため本書では割愛しました。興味がある方は調べてみてください。

Criteria API

JPQLを生成するAPIです。JPQLは文字列として組み立てるので、プログラムを実行するまで誤りがわかりません(コンパイラは文字列リテラルの内部をチェックしない)。これに対しCriteria APIは、メソッドやメタクラスというものを使いJPQLを作成することにより、エラーが起こらないようにしています(その代わりJPQLよりも複雑です)。

10.2 DAO(Data Access Object)

JPQL/Criteria APIによる動的クエリは、リポジトリを使わず自分で検索メソッドを作成します。Spring Bootを利用したアプリケーション開発では、リポジトリを使わずデータベースにアクセスする処理は**DAO(Data Access Object)**と呼ばれるクラスに記述するのが一般的です。本書でもこれにならってDAOのインターフェースと実装クラスを定義します。

以下がインターフェース定義です。

【リスト10-1】com.example.todolist.dao.TodoDao.java

```
package com.example.todolist.dao;

import java.util.List;
import com.example.todolist.entity.Todo;
import com.example.todolist.form.TodoQuery;

public interface TodoDao {
    // JPQLによる検索
    List<Todo> findByJPQL(TodoQuery todoQuery);

    // Criteria APIによる検索
    List<Todo> findByCriteria(TodoQuery todoQuery);
}
```

JPQL/Criteria API用の検索メソッドを宣言しています(実際に使うのは片方です)。どちらも検索条件がバインドされたTodoQueryオブジェクトを受け取り、検索結果をList<Todo>で返す仕様です。

> DAOという呼び名はクラスの役割を言い表したものです。実体はただのJavaクラスであり、何か特別な定義を求められているわけではありません。

10.2　JPQLによる動的クエリの実行

10.2.1 JPQLの組み立て

次にTodoDaoインターフェースを実装したクラスを作成します。

最初にJPQLで検索するfindByJPQL()について説明します。プログラムは以下のようになっています。

【リスト10-2】com.example.todolist.dao.TodoDaoImpl.java(一部抜粋)

```
package com.example.todolist.dao;

import java.util.ArrayList;
import java.util.List;
import com.example.todolist.common.Utils;
import com.example.todolist.entity.Todo;
import com.example.todolist.form.TodoQuery;
import jakarta.persistence.EntityManager;
import jakarta.persistence.Query;
import lombok.AllArgsConstructor;

@AllArgsConstructor
public class TodoDaoImpl implements TodoDao {
    private final EntityManager entityManager;

    // JPQLによる検索
    @Override
    public List<Todo> findByJPQL(TodoQuery todoQuery) {
        // ここを"todo"にすると実行時エラーになる。
        StringBuilder sb = new StringBuilder("select t from Todo t where
1 = 1");
        List<Object> params = new ArrayList<>();
        int pos = 0;
```

```
        // 実行するJPQLの組み立て
        // 件名
        if (todoQuery.getTitle().length() > 0) {
            sb.append(" and t.title like ?" + (++pos));          //  ①
            params.add("%" + todoQuery.getTitle() + "%");        //  ②
        }

        // 重要度
        if (todoQuery.getImportance() != -1) {
            sb.append(" and t.importance = ?" + (++pos));         //  ①
            params.add(todoQuery.getImportance());                //  ②
        }

        // 緊急度
        if (todoQuery.getUrgency() != -1) {
            sb.append(" and t.urgency = ?" + (++pos));           //  ①
            params.add(todoQuery.getUrgency());                   //  ②
        }

        // 期限：開始～
        if (!todoQuery.getDeadlineFrom().equals("")) {
            sb.append(" and t.deadline >= ?" + (++pos));             //  ①
            params.add(Utils.str2date(todoQuery.getDeadlineFrom())); //
②

        }

        // ～期限：終了で検索
        if (!todoQuery.getDeadlineTo().equals("")) {
            sb.append(" and t.deadline <= ?" + (++pos));                //
①

            params.add(Utils.str2date(todoQuery.getDeadlineTo()));  //  ②
        }

        // 完了
```

```
        if (todoQuery.getDone() != null &&
todoQuery.getDone().equals("Y")) {
            sb.append(" and t.done = ?" + (++pos));                  //  ①
            params.add(todoQuery.getDone());                         //  ②
        }

        // order
        sb.append(" order by id");

        Query query = entityManager.createQuery(sb.toString());    //  ③
        for (int i = 0; i < params.size(); ++i) {                    //  ④
            query = query.setParameter(i + 1, params.get(i));
        }

        @SuppressWarnings("unchecked")
        List<Todo> list = query.getResultList();                    //  ⑤
        return list;
    }

    // Criteria APIによる検索
    @Override
    public List<Todo> findByCriteria(TodoQuery todoQuery) {
        // 内容は次節で解説
        return null;
    }
}
```

処理の流れは、以下のようになっています。

①JPQLを文字列として組み立てる

②検索条件値(パラメータ)を保存

③①の結果からQueryオブジェクトを生成する

④②のパラメータをQueryオブジェクトにセットする

⑤Queryオブジェクトから検索結果を取得する

以下、各ステップの処理を説明します。

①JPQLを文字列として組み立てる

ここで作成するJPQLは、次のような形をしています(1=1の意味は後述します)。

```
select  t  from  Todo  t  where  1 =  1
                          and  TodoQueryオブジェクトから作成した検索条件
                    order  by  id
```

SELECT文そっくりですが、from句の"Todo"はTodoエンティティのことです。PostgreSQL上のtodoテーブルではありません。ただTodoエンティティは、@Tableでtodoテーブルと関連付けているため、最終的にはtodoテーブルに対する検索となります。

"Todo"の次の"t"はTodoのエイリアス(別名)です。これをselect句に書くとTodoの全プロパティを取得します(SELECT * と同じイメージ)。

またwhere句にプロパティ名を記述するときは、エイリアスで修飾します。

findByJPQL()で動的なのはwhere句の部分です。引数TodoQueryオブジェクトの全プロパティにパラメータがセットされていれば、次のようなJPQLを作成します。

```
select  t  from  Todo  t  where  1  =  1
                          and  t.title  like  ?1
                          and  t.importance  =  ?2
                          and  t.urgency  =  ?3
                          and  t.deadline  >=  ?4
                          and  t.deadline  <=  ?5
                          and  t.done  =  ?6
                    order  by  id
```

重要度と緊急度だけなら、次のようにします。

```
select  t  from  Todo  t  where  1  =  1
                          and  t.importance  =  ?1
                          and  t.urgency  =  ?2
                    order  by  id
```

JPQLに含まれる"?"で始まる数字(?1～?6)を「プレースホルダ」と言います。これは後続のステップ④でパラメータに置き換えられます。

しかしよく見ると、同じ?1でも最初の例ではtitle(件名)、次の例ではimportance(重要度)に使われています。このように各プレースホルダが何に対応するかは、パラメータの入力状態によって変わります。

> プレースホルダには、ここで使用した「?数値」の形式と、「:任意の文字列」の形式(たとえば:id)があります。前者を「位置パラメータ」、後者を「名前付きパラメータ」と言います。興味がある方は名前付きパラメータも調べてみてください。

なおJPQL中の"1 = 1"は、ちょっとしたハック(hack)です。これは真(true)になる条件であり、条件式の評価には影響しません(「真 and 条件式」の結果は、条件式の評価結果(true/false)になるため)。

そして"1 = 1"があると、後続の検索条件は常に"and "で始められます。つまり「最初の条件はandなし」「2つ目からはandをつける」といった判断ロジックが不要になる、というわけです。

②パラメータを保存

検索に使う値(パラメータ)をArrayListオブジェクトに保存します。④では、この保存した順番でプレースホルダを置換します。

③①の結果からQueryオブジェクトを生成する

①で作成したJPQLを引数にして、EntityManager#createQuery()でQueryオブジェクトを生成します。

Queryオブジェクトは、SQLのSELECT文に相当します。

EntityManagerは、その名の通りエンティティによる操作を制御するものであり、各種メソッドを提供しています。

このEntityManagerオブジェクトの取得方法については、次節で説明します。

```
Query query = entityManager.createQuery(sb.toString());   //  ③
```

④②のパラメータをQueryオブジェクトにセットする

生成したqueryに含まれるプレースホルダ(?1～)を、setParameter()でパラメータに置換します。第1引数はプレースホルダの数字部分(?1なら1、"?"は不要)、第2引数がパラ

メータです。

```
  for (int i = 0; i < params.size(); ++i) {                    //  ④
    query = query.setParameter(i + 1, params.get(i));
  }
```

⑤Queryオブジェクトから検索結果を取得する

ここで作成している検索条件は、結果が複数件(0～n件)になるものです。こういった場合、Query#getResultList()を呼び出し、検索を実行します。結果はコレクションとして受け取ります。

```
  List<Todo> list = query.getResultList();                     //  ⑤
```

> 検索結果が1件であるとわかっていれば、Query#getSingleResult()を使うこともできます。興味がある方は調べてみてください。

これでfindByJPQL()を呼び出すと、動的クエリの結果が返されます。
次は、このメソッドを実行するコントローラー側を説明します。

10.2.2 EntityManager

コントローラー側のポイントは、EntityManagerの取得方法です。

【リスト10-3】com.example.todolist.controller.TodoListController.java(一部抜粋)

```
package com.example.todolist.controller;
      :
@Controller
@RequiredArgsConstructor                    //  ②
public class TodoListController {
    private final HttpSession session;
    private final TodoRepository todoRepository;
    private final TodoService todoService;
    // Todolist5で追加
```

```
    @PersistenceContext                 //  ①
    private EntityManager entityManager;
    TodoDaoImpl todoDaoImpl;

    @PostConstruct                      //  ③
    public void init() {
        todoDaoImpl = new TodoDaoImpl(entityManager);
    }
      :
    // フォームに入力された条件でToDoを検索(Todolist4で追加，Todolist5で変
更)
    @PostMapping("/todo/query")
    public ModelAndView queryTodo(@ModelAttribute TodoQuery todoQuery,
                                  BindingResult result,
                                  ModelAndView mv) {
        mv.setViewName("todoList");

        List<Todo> todoList = null;
        if (todoService.isValid(todoQuery, result)) {
            // エラーがなければ検索
            // todoList = todoQueryService.query(todoQuery);
            // ↓
            // JPQLによる検索
            todoList = todoDaoImpl.findByJPQL(todoQuery);  //  ④
        }

        //mv.addObject("todoQuery", todoQuery);
        mv.addObject("todoList", todoList);

        return mv;
    }
      :
}
```

EntityManagerのインスタンスは**@PersistenceContextアノテーション**で取得します(①)。

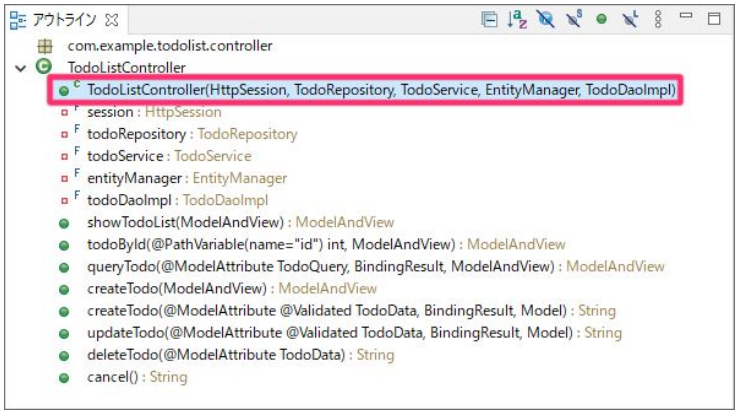
```
@PersistenceContext                    //  ①
private EntityManager entityManager;
```

> これはEntityManagerの作成タイミングが、@Autowiredでコンストラクターインジェクションするものとは異なるためです。興味がある方は調べてみてください。

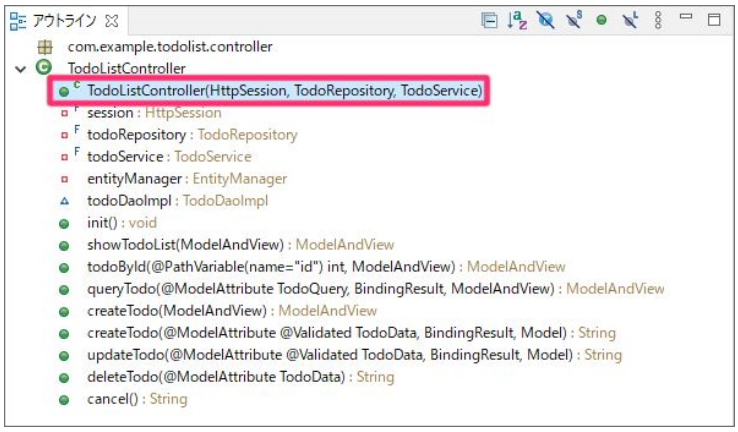
このためentityManagerをコンストラクターインジェクションの対象外とする必要があります。そこでコンストラクタ作成方法を@AllArgsConstructorから、**@RequiredArgsConstructor**へ変更します(②)。前者はクラスの全フィールドを対象としたコンストラクタを生成しますが、後者はfinalが指定されたフィールドだけになります。

両コンストラクタの違いはSTSのアウトラインで確認できます。

■@AllArgsConstructorの場合 - EntityManagerがコンストラクタに含まれている



■@RequiredArgsConstructorの場合 - EntityManagerが含まれていない



次のinit()には**@PostConstructアノテーション**を付与しています(③)。これは「init()の実行タイミングは、コンストラクタや@PersistenceContextによる初期化終了後」ということを表します。この結果TodoDaoImplのコンストラクタを経由してEntityManagerを渡せるようになります。

```
  @PostConstruct                          //  ③
  public void init() {
    todoDaoImpl = new TodoDaoImpl(entityManager);
  }
```

実際TodoDaoImpl には@AllArgsConstructorを付与しているので、コンストラクタ経由で受け取ります。

```
  @AllArgsConstructor
  public class TodoDaoImpl implements TodoDao {
    private final EntityManager entityManager;
```

そして**init()が終了してから**、queryTodo()などのメソッドが呼び出されます。

これでtodoDaoImplに対してfindByJPQL()を実行すれば、検索条件に応じたレコードを取得できるわけです(④)。

```
  todoList = todoDaoImpl.findByJPQL(todoQuery);  //  ④
```

addObject()からの処理は前章と同じです。同じ型のオブジェクトを渡しているので、一覧画面(todoList.html)側は変更する必要がありません。

10.4 Criteria APIによる動的クエリの実行

Criteria APIは複数のクラス、インターフェースが出てくるので複雑です。ここもステップ・バイ・ステップで、Criteria APIの基礎を解説した後、ToDoアプリへの適用例を説明します。

10.4.1 Criteria APIの基礎

Criteria APIで中心となるのは、次の3つのインターフェースです。

- **CriteriaBuilderインターフェース (jakarta.persistence.criteria.CriteriaBuilder)**
  Criteria APIによる検索を管理する。
- **CriteriaQueryインターフェース (jakarta.persistence.criteria.CriteriaQuery)**
  JPQLのselect句、from句、where句などに相当するものをメソッドで設定し、クエリを生成する。
- **Rootインターフェース(jakarta.persistence.criteria.Root)**
  エンティティの列に関する情報を表す。

これらのオブジェクトは次のように作成します。

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();
CriteriaQuery<Todo> query = builder.createQuery(Todo.class);
Root<Todo> root = query.from(Todo.class);
```

最初にEntityManager#getCriteriaBuilder()を呼び出してCriteriaBuilderを取得します。

次にCriteriaBuilder#createQuery()でCriteriaQueryを作成します。引数には検索で取得するエンティティクラスの「Classインスタンス」を渡します。Classインスタンスは、そのクラスで定義されているフィールドやメソッドなどの情報を保持しています。このTodo.classは、TodoのClassインスタンスなので、queryによる検索はTodoオブジェクトを返します。

RootはCriteriaQuery#from()で作成します。引数はTodoのClassインスタンスです。これでrootは、検索対象がTodoの全プロパティ、ということを表します。

以上が検索する準備です。ここから実際の検索処理に入っていきます。簡単なものから順を追って説明します。

全レコード取得する

最初は、最も単純な「todoテーブルの全レコードを取得する」ものです。これは

```
SELECT * FROM todo
```

に相当します。Criteria APIでは次のようにします。

```
List<Todo> list =
entityManager.createQuery(query.select(root)).getResultList();
```

最も内側のCriteriaQuery#select()は、取得するプロパティを指定するものです。前述のようにqueryは、検索対象エンティティがTodoであることを表しています。またrootはTodoの全プロパティを表しています。よってquery.select(root)は、todoテーブルの全列を取得対象にします。

EntityManager#createQuery()は、引数に従いTypedQuery型のクエリを作成します。ここではwhereに相当する検索条件がないので、todoテーブルの全レコード/全列を取得するクエリになります。

実際にクエリを実行するのは、TypedQuery#getResultList()です。検索結果はList<Todo>型に変換されて返されます。

検索条件を指定する

次は検索条件です。これにはCriteriaQuery#where()を使います。

たとえば「重要度が高(importance = 1)のToDo」を検索するとします。SELECT文で表せば、

```
SELECT * FROM todo WHERE importance = 1
```

です。これは次のようなコードになります。

```
query = query.select(root).where(builder.equal(root.get("importance"),
1));
List<Todo> list = entityManager.createQuery(query).getResultList();
```

select(),where()の戻り値はCriteriaQuery型です。このため上記のようにメソッドを続けて記述できます(メソッドチェイン)。そして結果は、再びCriteriaQuery型のqueryへ代入しています。

where()の引数は検索条件です。importance = 1という条件をCriteriaBuilder#equal()で作成します。

```
builder.equal(root.get("importance"), 1)
```

equal()は、第1引数のプロパティと第2引数の値が等しいこと(第1引数 = 第2引数)を検証するPredicateオブジェクトを作成します。つまりこのPredicateが検索条件になるわけです。

ここでは第1引数がimportanceプロパティを(=列)表しています。rootはTodoの全プロパティの情報を持っています。そこからRoot#get()でimportanceプロパティを取得し、第2引数の1と比較されるようにします。

ANDで複合条件にする

さらに「緊急度が高(urgency = 1)のToDo」という条件を追加します。SELECT文で表せば

```
SELECT * FROM todo WHERE importance = 1
                         AND urgency = 1
```

に相当します。これは次のようになります。

```
query = query
          .select(root)
          .where(
            builder.equal(root.get("importance"), 1),
            builder.and(builder.equal(root.get("urgency"), 1)));
List<Todo> list = entityManager.createQuery(query).getResultList();
```

where()の引数が2つになっています。where()はPredicateオブジェクトの可変個の引数を受け取るので、条件をカンマ(,)で区切り並べます。

equal()は前述のとおりです。ここでは新たにCriteriaBuilder#and()を使っています。これは上記where句と比べると、なんとなく働きがわかると思います。and()は、条件を論理積(AND)でつなぐものです。

もう１つ条件を追加します。期限が2020-10-10以前に設定されているToDoに限定します。SELECT文で表せば次のようになります。

```
SELECT * FROM todo WHERE importance = 1
```

```
                        AND  urgency  =  1
                        AND  deadline  <=  '2020-10-10'
```

以下がコーディング例です。

```
  query = query
          .select(root)
          .where(
            builder.equal(root.get("importance"), 1),
            builder.and(builder.equal(root.get("urgency"), 1),
            builder.and(builder.lessThanOrEqualTo(
                                root.get("deadline"),
                                Utils.str2date("2020-10-10")))));
  List<Todo> list = entityManager.createQuery(query).getResultList();
```

where()とand()については、前述のとおりです。

新しく出てきたのはCriteriaBuilder#lessThanOrEqualTo()です。これは第1引数のプロパティが第2引数の値より小さいか等しいこと(第1引数 <= 第2引数)を検証するPredicateオブジェクトを作成します。

このようにCriteriaBuilderには、演算子に対応するメソッドが定義されています。

なおUtils.str2date()は、前章で追加した「文字列をjava.sql.Dateオブジェクトへ変換する」メソッドです。deadlineプロパティはjava.sql.Date型なので、これに合わせる必要があるわけです。

動的生成への対応

ここまで説明してきたwhere()に検索条件を並べる方法はわかりやすいのですが、検索条件の動的生成には向いていません。なぜなら引数にする条件は、ユーザーが入力した内容で決まるからです。

そこで次のようにします。

```
  List<Predicate> predicates = new ArrayList<>();

  // 検索条件(Predicate)をListへ追加する
  predicates.add(builder.equal(root.get("importance"), 1));
  predicates.add(builder.and(builder.equal(root.get("urgency"), 1)));
  predicates.add(builder.and(builder.lessThanOrEqualTo(
                                    root.get("deadline"),
                                    Utils.str2date("2020-10-10"))));
```

```
    // Listを配列に変換する
    Predicate[] predArray = new Predicate[predicates.size()];
    predicates.toArray(predArray);

    // 配列を可変長引数としてwhere()渡す
    query = query.select(root).where(predArray);
    List<Todo> list = entityManager.createQuery(query).getResultList();
```

predicatesの実体はArrayList<Predicate>です。前述したように検索条件はPredicate型です。<>は左辺から<Predicate>と推論されます。

条件をpredicatesに追加し終わったら、Predicate型の配列へ変換し、where()に渡します。上述したようにwhere()はPredicateの可変長引数を受け取ります。そして可変長引数の実体は配列です。このためwhere()へ引数として配列を渡せるわけです。

これでユーザーが指定した条件をpredicatesへadd()すれば、動的なクエリとすることができます。

検索結果の並べ替え

最後は並べ替えです。前述の検索結果を「idの昇順(小→大)」で並べ替えるには、以下のようにします。

```
    query =

query.select(root).where(predArray).orderBy(builder.asc(root.get("id")))
;
```

CriteriaQuery#orderBy()は、並べ替えの指定です。これも戻り値がCriteriaQuery型です。

CriteriaBuilder#asc()は、引数のプロパティを昇順(小→大)にします。

root.get("id")はTodoのidプロパティを表すため、「idの昇順」になります。

以上がCriteria APIの簡単な解説です。次節ではこれを使ってToDoの動的クエリを実行します。

10.4.2 Criteria APIの利用

前節の内容をベースに作成したCriteria APIでToDoを検索するメソッドfindByCriteria()は、次のようになっています。

【リスト10-4】com.example.todolist.dao.TodoDaoImpl.java(一部抜粋)

```
package com.example.todolist.dao;

import java.util.ArrayList;
import java.util.List;
import com.example.todolist.common.Utils;
import com.example.todolist.entity.Todo;
import com.example.todolist.entity.Todo_;
import com.example.todolist.form.TodoQuery;
import jakarta.persistence.EntityManager;
import jakarta.persistence.Query;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Predicate;
import jakarta.persistence.criteria.Root;
import lombok.AllArgsConstructor;

@AllArgsConstructor
public class TodoDaoImpl implements TodoDao {
    private final EntityManager entityManager;
      :
    // Criteria APIによる検索
    @Override
    public List<Todo> findByCriteria(TodoQuery todoQuery) {
        CriteriaBuilder builder = entityManager.getCriteriaBuilder();
        CriteriaQuery<Todo> query = builder.createQuery(Todo.class);
        Root<Todo> root = query.from(Todo.class);
        List<Predicate> predicates = new ArrayList<>();

        // 件名
        String title = "";
```

```
        if (todoQuery.getTitle().length() > 0) {
            title = "%" + todoQuery.getTitle() + "%";
        } else {
            title = "%";
        }
        predicates.add(builder.like(root.get(Todo_.TITLE), title));

        // 重要度
        if (todoQuery.getImportance() != -1) {
          predicates.add(
            builder.and(
              builder.equal(
                root.get(Todo_.IMPORTANCE), todoQuery.getImportance())));
        }

        // 緊急度
        if (todoQuery.getUrgency() != -1) {
          predicates.add(
            builder.and(
              builder.equal(
                root.get(Todo_.URGENCY), todoQuery.getUrgency())));
        }

        // 期限：開始～
        if (!todoQuery.getDeadlineFrom().equals("")) {
          predicates.add(
            builder.and(
              builder.greaterThanOrEqualTo(
                root.get(Todo_.DEADLINE),
                Utils.str2date(todoQuery.getDeadlineFrom()))));
        }

        // ～期限：終了で検索
        if (!todoQuery.getDeadlineTo().equals("")) {
```

```
          predicates.add(
            builder.and(
              builder.lessThanOrEqualTo(
                root.get(Todo_.DEADLINE),
                Utils.str2date(todoQuery.getDeadlineTo()))));
        }

        // 完了
        if (todoQuery.getDone() != null &&
todoQuery.getDone().equals("Y")) {
          predicates.add(
            builder.and(
              builder.equal(
                root.get(Todo_.DONE), todoQuery.getDone())));
        }

        // SELECT作成
        Predicate[] predArray = new Predicate[predicates.size()];
        predicates.toArray(predArray);
        query = query

.select(root).where(predArray).orderBy(builder.asc(root.get(Todo_.id)));

        // 検索
        List<Todo> list =
entityManager.createQuery(query).getResultList();

        return list;
    }
}
```

コントローラー側を次のように変更すれば、Criteria APIでの検索に切り替えられます。

```
todoList = todoDaoImpl.findByJPQL(todoQuery);
```

```
↓
todoList = todoDaoImpl.findByCriteria(todoQuery);
```

このfindByCriteria()も引数のTodoQueryオブジェクトから検索条件を動的に組み立てます。もしパラメータがすべてセットされていれば、次のようなWHERE句に相当するものを作成します。

```
WHERE  title  LIKE  '%' + フォーム.件名 + '%'
  AND  importance  =  フォーム.重要度
  AND  urgency     =  フォーム.緊急度
  AND  deadline    >=  フォーム.期限：開始
  AND  deadline    <=  フォーム.期限：終了
  AND  done        = 'Y'
```

重要度と緊急度だけなら次のようになります。

```
WHERE  title  LIKE  '%'
  AND  importance  =  フォーム.重要度
  AND  urgency     =  フォーム.緊急度
```

2つめの例では、件名が入力されていないのに「title LIKE '%'」を条件にしています。意味としては「title列に何らかの文字が含まれる」ということであり、全レコード該当します。検索条件としては無意味ですが、これがあると後続の検索条件をすべてand()で作成できます(前述の1=1同様、一種のハックです)

ここで新しく出てきたメソッドは以下の2つです。

CriteriaBuilder#like()

・第1引数のプロパティが第2引数のパターンを満たすかどうか検証するPredicateオブジェクトを作成する。

・ワイルドカードは'%'で表す。

(第1引数 LIKE 第2引数)

CriteriaBuilder#greaterThanOrEqualTo()

・第1引数のプロパティが第2引数の値より大きいか等しいことを検証するPredicateオブジェクトを作成する。

(第1引数 >= 第2引数)

また前節のコードと違い、Root#get()の引数が"id"などの文字列ではなく、Todo_.IDといった形になっています。このTodo_はTodoの「メタクラス」です。メタクラスは「クラスの情報を持つクラス」であり、自動生成できます。以下は自動生成されたTodo_です。

【リスト10-5】com.example.todolist.entity.Todo_.java(自動生成結果)

```
package com.example.todolist.entity;

import java.sql.Date;
import jakarta.annotation.Generated;
import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.StaticMetamodel;

@Generated(value =
"org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Todo.class)
public abstract class Todo_ {

    public static volatile SingularAttribute<Todo, Integer> urgency;
    public static volatile SingularAttribute<Todo, Integer> importance;
    public static volatile SingularAttribute<Todo, Integer> id;
    public static volatile SingularAttribute<Todo, String> title;
    public static volatile SingularAttribute<Todo, Date> deadline;
    public static volatile SingularAttribute<Todo, String> done;

    public static final String URGENCY = "urgency";
    public static final String IMPORTANCE = "importance";
    public static final String ID = "id";
    public static final String TITLE = "title";
    public static final String DEADLINE = "deadline";
    public static final String DONE = "done";

}
```

メタクラスの自動生成方法は、本章の最後にありますので、そちらを参照してください。

メタクラスの詳細は割愛しますが、この中にはTodoエンティティのプロパティ名を表すフィールドが定義されています。これを使えばroot.get("importance")はroot.get(Todo_.IMPORTANCE)で表せます。root.get(Todo_.INPORTANCE)と打ち間違えても文法エラーになります。しかし文字列の場合は、root.get("inportance")としても実行時エラー発生まで気づけないでしょう。

またLikeなどSQLの演算子もメソッド化されているので、名称や引数の誤りはコンパイルエラーなります。Criteria APIを使うと、こういった誤りをコンパイル時点で見つけられます。

JPQLは文字列の組み立てが中心でわかりやすいのですが、文字列中に誤りがあってもコンパイルエラーになりません。
一方Criteria APIは、コンパイルで誤りを検出できますが、JPQLよりも複雑です。
このように一長一短があります。プロジェクトの約束事として、どちらを使うか(あるいは別な方法にするか)明文化されていることも多いと思います。基本的にはそれに従うべきでしょう。

> 本書で説明したJPQL, Criteria APIの機能は全体のごく一部です。詳細は以下のサイトなどを参照してください。
>
> ■JPQL
> 39.5 Full Query Language Syntax - Java Platform, Enterprise Edition: The Java EE Tutorial (Release 7)
> https://docs.oracle.com/javaee/7/tutorial/persistence-querylanguage005.htm
>
> ■Criteria API
> CriteriaBuilder (Java(TM) EE 7 Specification APIs)
> https://docs.oracle.com/javaee/7/api/javax/persistence/criteria/CriteriaBuilder.html

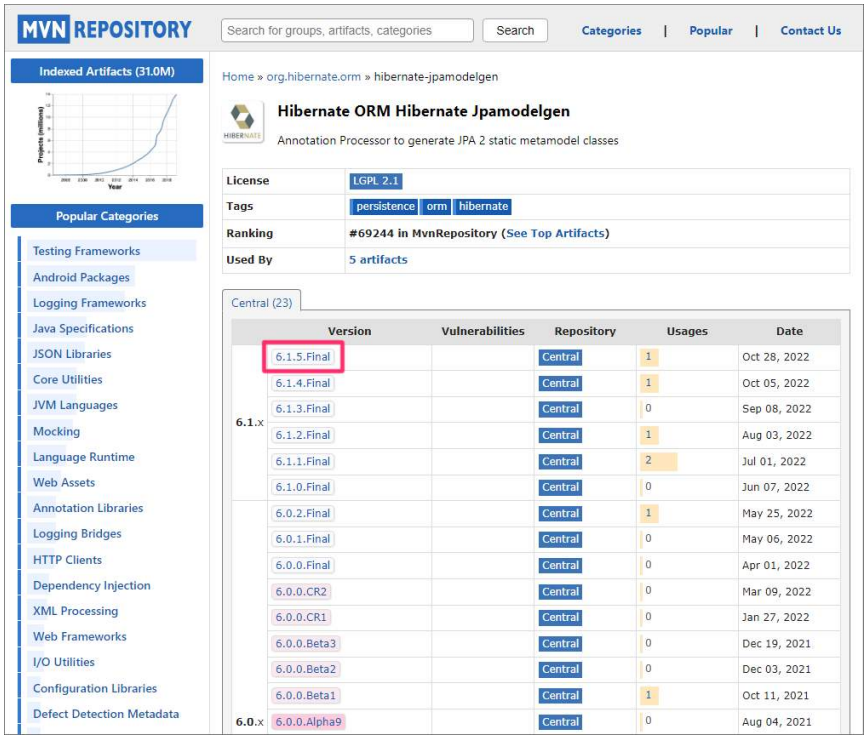
> CriteriaQuery (Java(TM) EE 7 Specification APIs)
> https://docs.oracle.com/javaee/7/api/javax/persistence/criteria/CriteriaQuery.html
> Root (Java(TM) EE 7 Specification APIs)
> https://docs.oracle.com/javaee/7/api/javax/persistence/criteria/Root.html

メタクラス作成手順

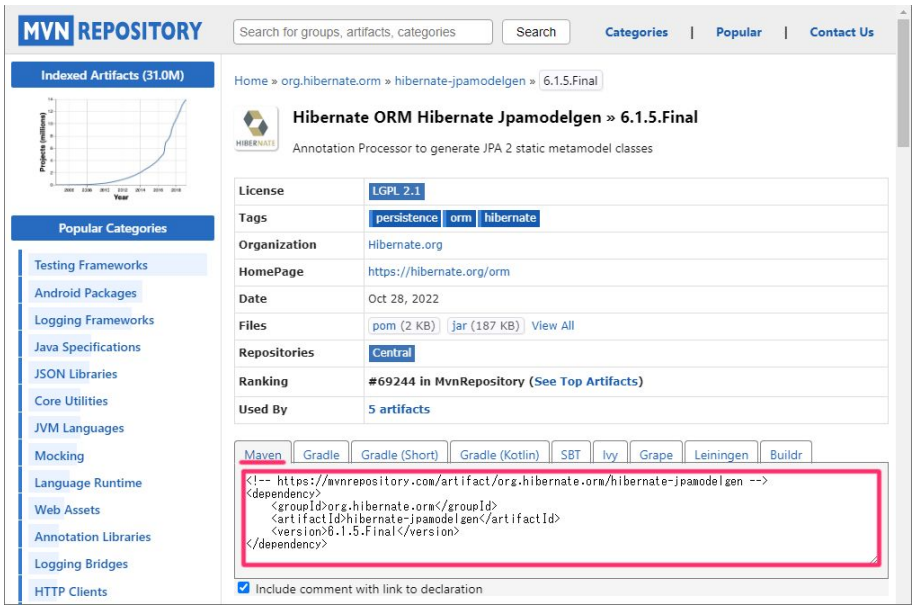
メタクラスの作成にはHibernate jpamodelgenというツールを使います。以下はそのインストール手順です。

1)ブラウザで
https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-jpamodelgenを開く。

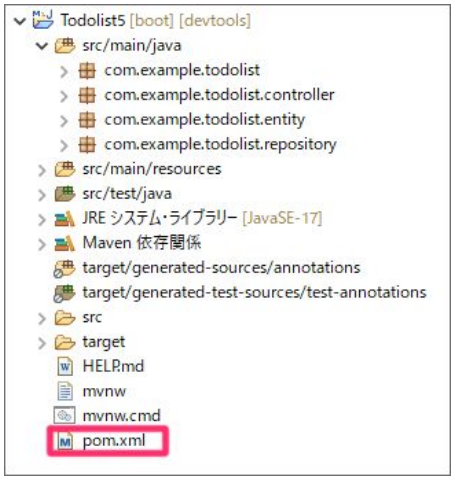
2)使用するVersionを選択する。
⇒ここでは6.1.5.Finalをクリック



3)Mavenタブの内容をコピーする。
⇒マウスで右クリックするとクリップボードにコピーされる。



4)STSのパッケージ・エクスプローラーでTodolist5のpom.xmlをダブルクリックして開く。



5)</dependencies>の直前に4)でコピーした内容を貼り付けて保存する([CTRL]+S)。

このタイミングでjpamodelgenのjarファイルがダウンロードされる。

■格納場所

C:\Users\<ユーザー名>\.m2\repository\org\hibernate\orm\hibernate-jpamodelgen\6.1.5.Final

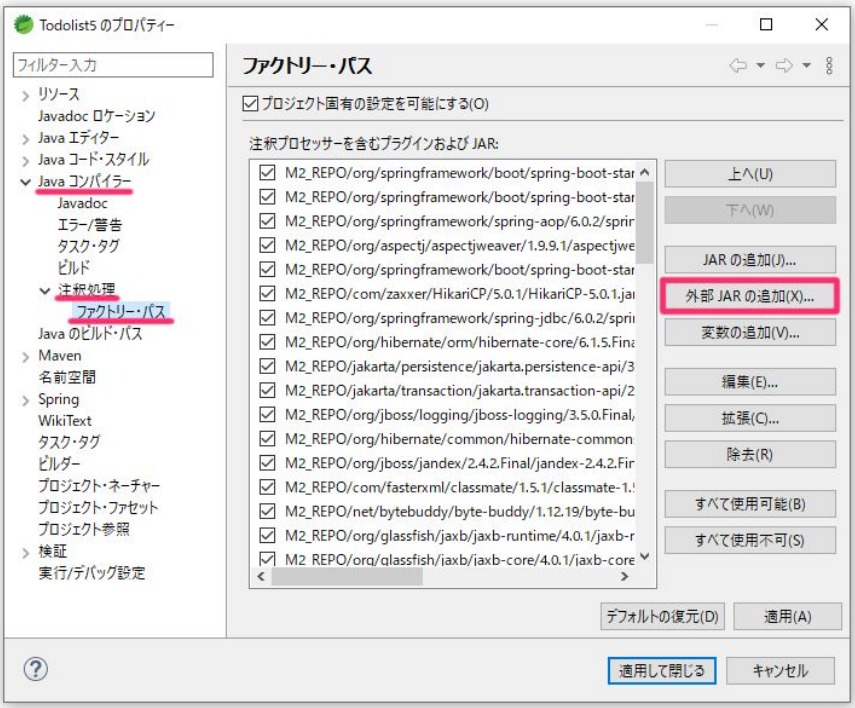
6)オプション：<version>～<version>の警告を消す

黄色波線にマウスカーソルを合わせる > クイックスフィックス Ignore this warningを選択

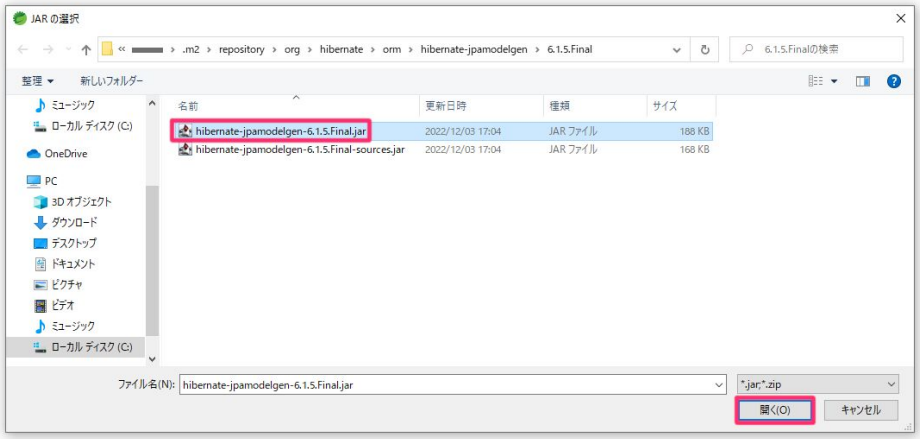
↓

7)パッケージ・エクスプローラーでTodolist5を右クリック > [プロパティ(R)] > [Javaコンパイラー] > [注釈処理]
> [ファクトリー・パス]を選択する。

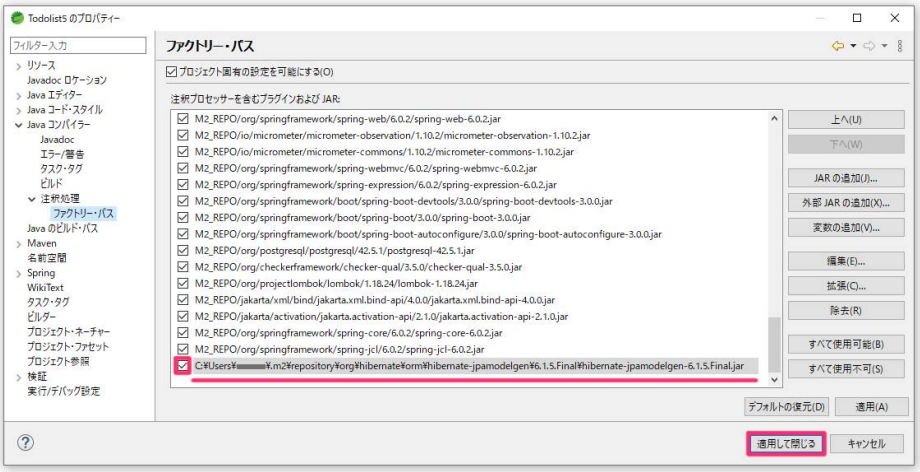
8)[外部JARの追加(X)...]ボタンをクリックする。



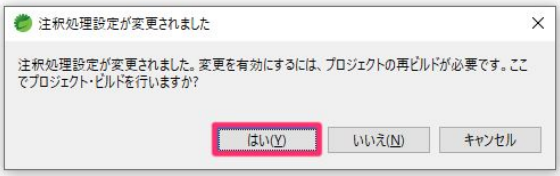
9)5)の格納場所にあるhibernate-jpamodelgen-6.1.5.Final.jarを選択し[開く(O)]ボタンをクリックする。



10)[注釈プロセッサーを含むプラグインおよびJAR]に9)で選択したjarファイルがチェックされていることを確認して[適用して閉じる]ボタンをクリックする。

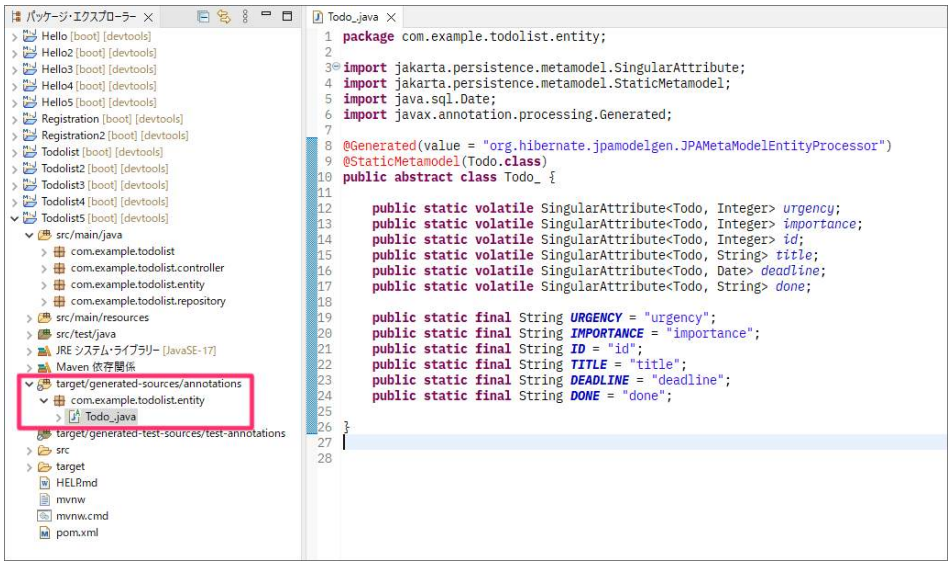


11)「注釈処理設定が変更されました」ダイアログに対して[はい(Y)]ボタンをクリックする。



12)エンティティクラスのメタデータクラスが作成される。

⇒ただし`src/main/java`ではなく`target/generated-sources/annotations`下に作成されます。



これ以降エンティクラス(@Entityを含むクラス)を作成したり、変更するとその内容に応じてメタクラスが自動的に作成/変更されます。手動で操作する必要はありません。

11．ページネーション(ページング)

検索処理に欠かせない機能として「ページネーション(ページング)」があります。これは検索結果一覧など、行数の多いデータを複数のWebページに分割し、各ページへのリンクを並べてアクセスしやすくするものです。

Spring BootではPageable/Pageを利用することで、ページネーションを簡単に実現できます。

Pageableインターフェース(org.springframework.data.domain.Pageable)

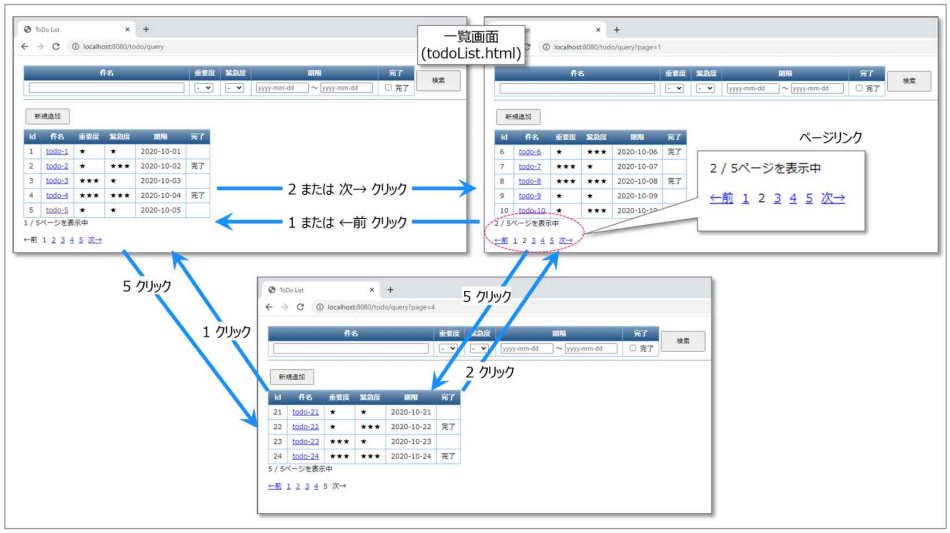
・ページネーションのためのインターフェース

・Pageに関するデータを管理、取得する機能を提供する。

Pageインターフェース(org.springframework.data.domain.Page)

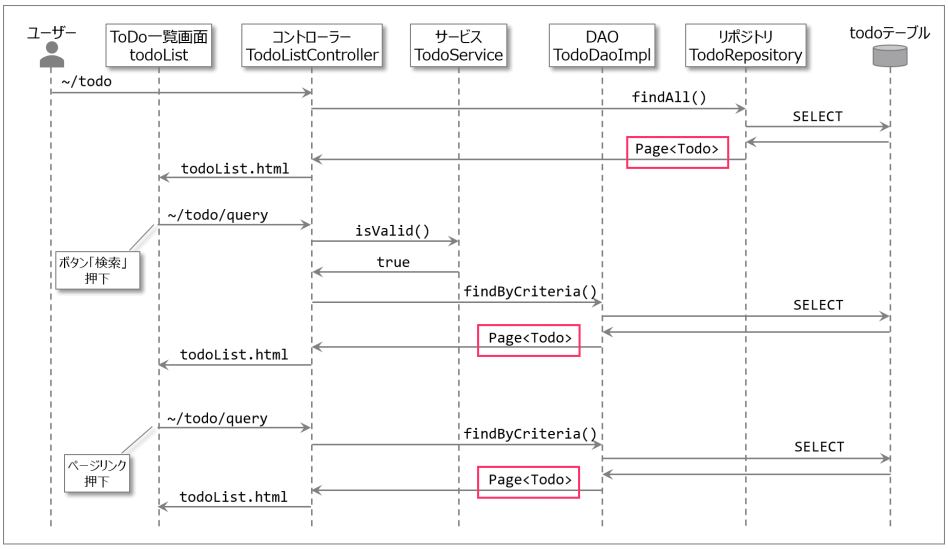
・検索結果のサブリスト

・検索結果における当該ページの位置情報を取得する機能などを提供する。



【図11-1】ToDoのページング処理

シーケンス図で表すと、次のようになります。findAll()および前章で作成したfindByCriteria()の戻り値がList<Todo>からpage<Todo>に変わっています。つまりページングに関する処理は、ここで実行しています。



【図11-2】ページング処理のシーケンス

作成するプロジェクトの仕様

プロジェクト名	Todolist6
依存関係	Spring Web, Spring Boot DevTools, Thymeleaf, Lombok, Spring Data JPA, PostgreSQL Driver, Validation

```
Todolist6
├ src/main/java
│  ├ com.example.todolist
│  │  └ TodolistApplication.java
│  ├ com.example.todolist.common
│  │  └ Utils.java
│  ├ com.example.todolist.controller
│  │  └ TodoListController.java(▲)
│  ├ com.example.todolist.dao
│  │  ├ TodoDao.java(▲)
│  │  └ TodoDaoImpl.java(▲)
│  ├ com.example.todolist.entity
│  │  ├ Todo.java
│  │  └ Todo_.java
│  ├ com.example.todolist.form
```

```
│  │  ├ TodoData.java
│  │  └ TodoQuery.java(▲)
│  ├ com.example.todolist.repository
│  │  └ TodoRepository.java
│  └ com.example.todolist.service
│      └ TodoService.java
└ src/main/resources
   ├ static
   │  └ css
   │     └ style.css
   └ templates
      ├ todoForm.html
      └ todoList.html(▲)
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

ページング処理されている様子がわかりやすくなるように、todoテーブルを一度初期化してレコードを増やします。以下のSQLファイルを、6章の手順を参考にしてpsqlから実行してください(接続ユーザーはtodouserです)。

【リスト11-1】init_table_2.sql

```
DROP TABLE todo;
CREATE TABLE todo
(
  id           SERIAL PRIMARY KEY,
  title        TEXT,
  importance    INTEGER,
  urgency      INTEGER,
  deadline     DATE,
  done         TEXT
);

INSERT INTO todo(title,importance,urgency,deadline,done)
VALUES('todo-1',0,0,'2020-10-01','N');
INSERT INTO todo(title,importance,urgency,deadline,done)
```

```
VALUES('todo-2',0,1,'2020-10-02','Y');
      :
INSERT INTO todo(title,importance,urgency,deadline,done)
VALUES('todo-23',1,0,'2020-10-23','N');
INSERT INTO todo(title,importance,urgency,deadline,done)
VALUES('todo-24',1,1,'2020-10-24','Y');
```

この状態でTodolist5(あるいはTodolist5からコピーした直後のTodolist6)を実行すると次のようになります。

id	件名	重要度	緊急度	期限	完了
1	todo-1	★	★	2020-10-01	
2	todo-2	★	★★★	2020-10-02	完了
3	todo-3	★★★	★	2020-10-03	
4	todo-4	★★★	★★★	2020-10-04	完了
5	todo-5	★	★	2020-10-05	
6	todo-6	★	★★★	2020-10-06	完了
7	todo-7	★★★	★	2020-10-07	
8	todo-8	★★★	★★★	2020-10-08	完了
9	todo-9	★	★	2020-10-09	
10	todo-10	★	★★★	2020-10-10	完了
11	todo-11	★★★	★	2020-10-11	
12	todo-12	★★★	★★★	2020-10-12	完了
13	todo-13	★	★	2020-10-13	
14	todo-14	★	★★★	2020-10-14	完了
15	todo-15	★★★	★	2020-10-15	
16	todo-16	★★★	★★★	2020-10-16	完了
17	todo-17	★	★	2020-10-17	
18	todo-18	★	★★★	2020-10-18	完了
19	todo-19	★★★	★	2020-10-19	
20	todo-20	★★★	★★★	2020-10-20	完了
21	todo-21	★	★	2020-10-21	
22	todo-22	★	★★★	2020-10-22	完了
23	todo-23	★★★	★	2020-10-23	
24	todo-24	★★★	★★★	2020-10-24	完了

これをページングできるようにしていきます。

11.1 Pageable/Pageの追加

ページングするには、以下の情報をコントローラークラスで作成し、ページリンクを表示する画面へ渡す必要があります。

1)指定されたページのコンテンツ(本章の場合、1ページ分のToDoのリスト)

2)表示するページに関する情報(ページ番号、1ページ当たりの表示件数、など)

本章ではToDo一覧画面(todoList.html)をページングしますが、この画面を表示する経路は次のように3つあります(シーケンス図も参照)。

【ルート1】http://localhost:8080/todoをアクセスして表示する場合

【ルート2】[検索]ボタンをクリックして表示する場合

【ルート3】ページリンクをクリックして表示する場合

つまりそれぞれのルートで、上記のページ単位の切り出し、ページ情報作成を行います。

最初に【ルート1】をステップ・バイ・ステップで作成します。その後、【ルート2】【ルート3】にも適用していきます。

まず【ルート1】に対応する@GetMapping("/todo")のハンドラーメソッドを次のように変更します。

【リスト11-2】

com.example.todolist.controller.TodoListController.java(step1;一部抜粋)

```
  @GetMapping("/todo")
  public ModelAndView showTodoList(ModelAndView mv, Pageable pageable) { //
①
      mv.setViewName("todoList");

      Page<Todo> todoList = todoRepository.findAll(pageable);       // ②, ③
      mv.addObject("todoQuery", new TodoQuery());
      mv.addObject("todoList", todoList);

      return mv;
```

```
    }
```

変更したのは次の3か所です。

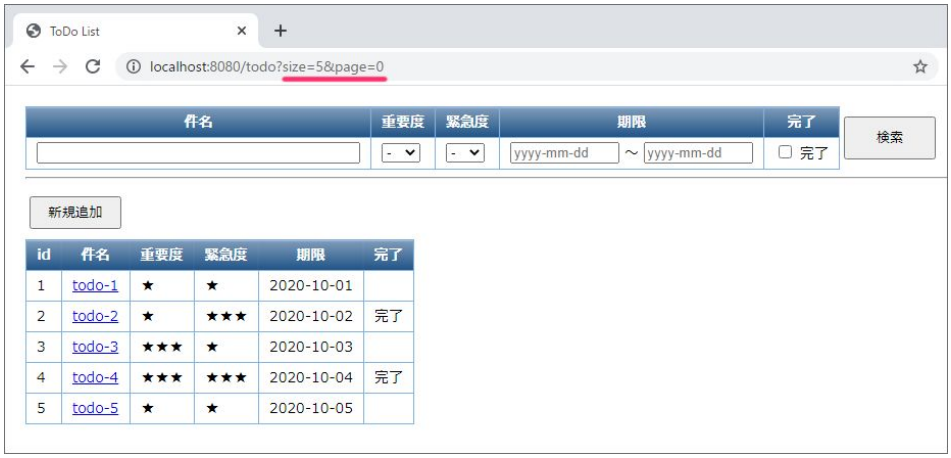
①メソッドの引数にPageableオブジェクトを追加

②findAll()に①のPageableオブジェクトを引数として渡す。

⇒このfindAll(Pageable pageable)も【表6-5】にあります。つまり自動実装されています。

③findAll()の結果を代入するtodoListを、②に合わせてList<Todo>からPage<Todo>に変更する。

これでTodolist6を起動し、ブラウザからhttp://localhost:8080/todo?size=5&page=0を開くと、最初の5件が表示されます。



id	件名	重要度	緊急度	期限	完了
1	todo-1	★	★	2020-10-01	
2	todo-2	★	★★★	2020-10-02	完了
3	todo-3	★★★	★	2020-10-03	
4	todo-4	★★★	★★★	2020-10-04	完了
5	todo-5	★	★	2020-10-05	

http://localhost:8080/todo?size=5&page=1にすると、次の5件が表示されます。

id	件名	重要度	緊急度	期限	完了
6	todo-6	★	★★★	2020-10-06	完了
7	todo-7	★★★	★	2020-10-07	
8	todo-8	★★★	★★★	2020-10-08	完了
9	todo-9	★	★	2020-10-09	
10	todo-10	★	★★★	2020-10-10	完了

http://localhost:8080/todo?size=5&page=0&sort=id,DESCでは、id順で最後から5件表示されます。

【リスト11-2】だけで、ToDoをページングできました。実行結果からある程度わかると思いますが、ハンドラーメソッドの引数にPageableオブジェクトがあると、リクエストパラメータsize，page，sortは、下表のような意味を持ちます。これらが【リスト11-2】①のPageableに取り込まれ、それを②findAll()に渡すことで、指定した部分(ページ)を取得できるようになります。

【表11-1】ページングのパラメータ

パラメータ名	説明
size	・ページ当たりの表示件数(省略時:20)
page	・表示するページ番号(省略時:0) ・先頭ページは0，2ページ目は1を指定する。
sort	・ソート条件(複数指定可能,省略時:UNSORTED) ・条件は"ソート項目名(,ソート順)" で指定する。 ・ソート順は昇順(小→大)なら"ASC"，降順(大→小)なら"DESC"とする(省略時:"ASC") 例."sort=x,y,DESC&sort=z" → "ORDER BY x DESC，y DESC，z ASC"と解釈する。

※上表のパラメータは、@RequestParamで取得しないことに注意

これでページ単位の表示ができるようになりました。しかし毎回これらのパラメータを指定するのは面倒です。そこで一般的なWebサイトのように、ページリンクで移動できるよ

うにします。

【リスト11-3】

com.example.todolist.controller.TodoListController.java(step2;一部抜粋)

```
  @GetMapping("/todo")
  public ModelAndView showTodoList(ModelAndView mv,
      @PageableDefault(page = 0, size = 5, sort = "id") Pageable
pageable) {//①
      mv.setViewName("todoList");

      Page<Todo> todoPage = todoRepository.findAll(pageable);
      mv.addObject("todoQuery", new TodoQuery());
      mv.addObject("todoPage", todoPage);                    //  ②
      mv.addObject("todoList", todoPage.getContent());      //  ③
      session.setAttribute("todoQuery", new TodoQuery());   //  ④

      return mv;
  }
```

【リスト11-2】から次の4か所を変更します。

①引数のPageableオブジェクトに**@PageableDefaultアノテーション**を追加

size, page, sortのデフォルト値を指定します。

②findAll()の実行結果であるPage<Todo>オブジェクトを一覧画面に渡す処理を追加

このオブジェクトを使って一覧画面(todoList.html)側でページリンクを作成します(次節)。

③表示内容をgetContent()で取得

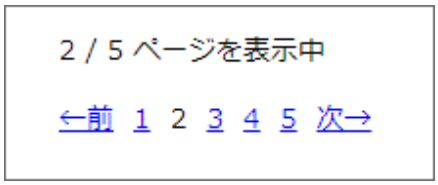
Pageオブジェクトはページ情報のほかに、(size,page,sortから求めた)次に表示するページ単位のデータ(コンテンツ)を持っています。これをgetContent()で取得し、一覧画面に渡します。

④セッションに検索条件を格納する

ページリンクで次に表示ページを検索するときに備えて、現在の検索条件(=条件・無)をセッションへ格納しておきます。

11.2 ページリンクの作成

一覧画面では、コントローラーから渡されたPageオブジェクトを使ってページリンクを作成します。本章のページリンクは、以下のような形式です。



1行目：現在表示しているページ位置
2行目：各ページへのリンク(表示中のページはリンクにしない)

画面もステップ・バイ・ステップで変更していきます。

【リスト11-4】src/main/resources/templates/todoList.html(step1;一部抜粋)

```
  :
 </table>
 <div>
   <span th:text="|${todoPage.getNumber() + 1} / ${todoPage.getTotalPages()} ページを表示中|">
   </span>
   <ul id="nav">
     <li>
       <span th:if="${todoPage.isFirst()}">←前</span>
       <a th:unless="${todoPage.isFirst()}"
          th:href="@{/todo/query(page = ${todoPage.getNumber() - 1})}">←前</a>
     </li>
     <li th:each="i : ${#numbers.sequence(0, todoPage.getTotalPages() - 1)}">
       <span th:if="${i == todoPage.getNumber()}" th:text="${i + 1}"></span>
         <a th:if="${i != todoPage.getNumber()}"
            th:href="@{/todo/query(page=${i})}" th:text="${i + 1}"></a>
     </li>
     <li>
       <span th:if="${todoPage.isLast()}">次→</span>
       <a th:unless="${todoPage.isLast()}"
```

```
            th:href="@{/todo/query(page = (${todoPage.getNumber()+ 1}))}">次→</a>
        </li>
      </ul>
    </div>
</body>
</html>
```

todoPageは上記②でコントローラーから渡されたPageオブジェクトです。Pageオブジェクトからは、下表のメソッドで各種情報を取得できます。これを利用してページリンクを作成します。

【表11-2】Pageインターフェースのメソッド

メソッド	機能
getNumber()	表示しているページの番号を返す(先頭:0)
getTotalPages()	総ページ数を返す。
isFirst()	先頭ページを表示しているときtrueを返す。
isLast()	最終ページを表示しているときtrueを返す。

最初のspan要素は「表示中ページ/総ページ数」です。

```
<span th:text="|${todoPage.getNumber() + 1} / ${todoPage.getTotalPages()} ページを表示中|"></span>
```

表示中のページ番号はgetNumber()で取得できますが、0から始まるので+1します。

総ページ数はgetTotalPages()です。

ここではth:text="| ... |"というように|で囲まれている部分があります。この記法を使うとテキスト内に変数式${...}を埋め込めるので、+ で結合するよりシンプルになります。

最初のli要素は「←前」です。ここはisFirst()で「先頭ページかどうか？」を判定し、以下のように表示を変えます。

・先頭ページの場合、単なる文字列とする(リンクにしない)。

・先頭ページでなければ、前ページへのリンクとする。

```
<span th:if="${todoPage.isFirst()}">←前</span>
```

```
<a th:unless="${todoPage.isFirst()}"
   th:href="@{/todo/query(page = ${todoPage.getNumber() - 1})}">←前</a>
```

同じ条件をth:ifとth:unlessで評価しているので、どちらかが作成されます。このうちth:unlessには、URLリンク式に(page = ${todoPage.getNumber() - 1})という()で囲まれた部分があります。これはクエリ文字列を生成する記法です。もし表示するのが3ページ目であれば、次のようになります。

```
th:href="@{/todo/query(page = ${todoPage.getNumber() - 1})}"
↓
th:href="@{/todo/query(page = ${2 - 1})}"
↓
th:href="@{/todo/query(page = 1)}"
↓
href="/todo/query?page=1"
```

２つの目li要素は各ページへのリンクです。

```
<li th:each="i : ${#numbers.sequence(0, todoPage.getTotalPages() - 1)}">
```

#numbersはThymeleafの数値用ユーティリティオブジェクトです。sequence()は引数の範囲の配列を生成します。これで「0 ～ 総ページ数-1」を要素とする配列を作り、それをiに代入しながら子要素を繰り返します。つまり全ページ番号分の処理を行うわけです。

ここでも以下のように、表示を変えます。

・iが表示するページの場合、文字列として表示する(リンクにしない)。
・表示ページでなければ、そのページへのリンクとする。

この「表示ページ」はgetNumber()で取得します。

```
<span th:if="${i == todoPage.getNumber()}" th:text="${i + 1}"></span>
<a th:unless="${i == todoPage.getNumber()}"
   th:href="@{/todo/query(page=${i})}" th:text="${i + 1}"></a>
```

最後のli要素は「次→」の部分です。最終ページの判定はisLast()で行います。考え方は「←前」と同じです。

これで【ルート1】のページングは完了です。次は【ルート2】の[検索]ボタンクリックで動的クエリを実行した場合です。

11.3 動的クエリ結果のページング

[検索]ボタンクリックに対応するハンドラーメソッドを、以下のように変更します。

【リスト11-5】

com.example.todolist.controller.TodoListController.java(step3;一部抜粋)

```
  @PostMapping("/todo/query")
  public ModelAndView queryTodo(@ModelAttribute TodoQuery todoQuery,
                                BindingResult result,
                                @PageableDefault(page = 0, size = 5) Pageable
pageable, // ①
                                ModelAndView mv) {
      mv.setViewName("todoList");

      Page<Todo> todoPage = null;                                  //  ②
      if (todoService.isValid(todoQuery, result)) {
          // エラーがなければ検索
          todoPage = todoDaoImpl.findByCriteria(todoQuery, pageable);    //  ③

          // 入力された検索条件をsessionに保存
          session.setAttribute("todoQuery", todoQuery);      //  ④

          mv.addObject("todoPage", todoPage);                //  ⑤
          mv.addObject("todoList", todoPage.getContent()); //  ⑥

      } else {
          // エラーがあった場合検索
          mv.addObject("todoPage", null);                     //  ⑤'
          mv.addObject("todoList", null);                     //  ⑥'
      }

    return mv;
  }
```

前章から追加、変更したのは以下の部分です。

①メソッド引数にPageableオブジェクトと初期値を追加

②検索結果の型をList<Todo>からPage<Todo>に変更

③(前章で作成した)findByCriteria()の引数にPageableオブジェクトを追加

⇒ページ単位の検索結果とする(後述)

④検索条件をセッションに保存する処理を追加

⇒ページリンクはGETリクエストになるのでフォームの検索条件をサーバーへ送信できない。

⇒そのためこの時点(=[検索]ボタンクリック時)でバインドされた条件をセッションに格納し、後続画面で使う。

⑤ページ情報を画面に渡す。

⑥表示する検索結果を画面に渡す。

⑤'、⑥'検索条件にエラーがあった場合は、ページ情報、検索結果にnullをセットして画面に渡す。

エラーチェックを除けば、基本的な流れは前述のshowTodoList()と同じです。これに沿って、メソッドを変更していきます。

まずfindByCriteria()にPageableオブジェクトを渡せるようインターフェースを変更します。

【リスト11-6】com.example.todolist.dao.TodoDao.java

```
package com.example.todolist.dao;

import java.util.List;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import com.example.todolist.entity.Todo;
import com.example.todolist.form.TodoQuery;

public interface TodoDao {
    // JPQLによる検索
    List<Todo> findByJPQL(TodoQuery todoQuery);

    // Criteriaによる検索(Todolist6でPageable追加)
```

```
    Page<Todo> findByCriteria(TodoQuery todoQuery, Pageable pageable);
}
```

次に実装を変更します。クエリを作成するところまでは、前章と同じです。戻り値がPageableで指定された部分だけとなるようにします。

【リスト11-7】com.example.todolist.dao.TodoDaoImpl.java(一部抜粋)

```
  @Override
  public Page<Todo> findByCriteria(TodoQuery todoQuery, Pageable pageable) {
    :
    // SELECT作成
    Predicate[] predArray = new Predicate[predicates.size()];
    predicates.toArray(predArray);
    query =
query.select(root).where(predArray).orderBy(builder.asc(root.get(Todo_.id)));

    // クエリ生成
    TypedQuery<Todo> typedQuery = entityManager.createQuery(query);  //  ①
    // 該当レコード数取得
    int totalRows = typedQuery.getResultList().size();               //  ②
    // 先頭レコードの位置設定
    typedQuery.setFirstResult(pageable.getPageNumber() * pageable.getPageSize());
//  ③
    // 1ページ当たりの件数
    typedQuery.setMaxResults(pageable.getPageSize());                //  ④

    Page<Todo> page = new PageImpl<Todo>(typedQuery.getResultList(), pageable,
totalRows); //⑤
    return page;                                                      //  ⑥
  }
```

ポイントは最後にPage<Todo>オブジェクトをreturnしているところです(⑥)。

先に説明したshowTodoList()で呼び出したfindAll(Pageable pageable)の戻り値もPage<Todo>でした。そしてこのオブジェクトから表示するデータを取得し、さらに画面側

でページリンクを作成しました。findByCriteria()の結果もページングするには、Page<Todo>で返す必要があるわけです。

Page<T>インターフェースには、デフォルト実装クラスとしてPageImpl<T>があります。ここでは、以下のコンストラクタで生成します。

```
PageImpl(List<T> content, Pageable pageable, long total)
・content - 検索結果(該当ページ分)
・pageable - ページング情報
・total - 検索結果の件数
```

contentは検索結果全体ではなく、pageableで指定されたページ分だけとします。たとえば「11件目からの5件分」という形です。このページ単位の取得には、クエリの実行を制御するTypedQueryインターフェースを使ってみます。そのためcreateQuery()の結果を取得します(①)。

```
List<Todo> list = entityManager.createQuery(query).getResultList();
↓
TypedQuery<Todo> typedQuery = entityManager.createQuery(query);   //  ①
```

前章で説明したようにEntityManager#createQuery()の戻り値はTypedQuery型です。

②のgetResultList()は検索結果をListオブジェクトにして返します。この要素数をsize()で取得して、条件に該当したレコード数を求めます。

次に検索結果のどの位置から何件取り出すかを、setFirstResult()およびsetMaxResults()で設定します。

開始位置は、引数で渡されたpageableの

　表示ページ番号(=getPageNumber()) * 1ページ当たりの表示件数(=getPageSize())

で算出できます(③)。

取得件数はgetPageSize()そのものです(④)。

これでgetResultList()を実行すると、setFirstResult()で指定した位置からsetMaxResults()の件数分の検索結果を得られます(⑤)。あとはこの結果をリターンすればよいわけです(⑥)。

```
Page<Todo> page = new PageImpl<Todo>(typedQuery.getResultList(), pageable,
totalRows);  //  ⑤
```

> ここではidの昇順で並べ替えた結果をページングしています。order byが無いと意図しない結果になる可能性があるので注意してください。

画面側のページリンクは、本章の前半で作成したものに2か所修正を加えます。

【リスト11-8】src/main/resources/templates/todoList.html(step2;一部抜粋)

```
<div th:if="${todoList != null && #lists.size(todoList) != 0}"> <!-- ① -->
  <span th:text="|${todoPage.getNumber() + 1} / ${todoPage.getTotalPages()} ページを表示中|">
  </span>
  <ul id="nav">
    <li>
      <span th:if="${todoPage.isFirst()}">←前</span>
      <a th:unless="${todoPage.isFirst()}"
         th:href="@{/todo/query(page = ${todoPage.getNumber() - 1})}">←前</a><!-- ② -->
    </li>
    <li th:each="i : ${#numbers.sequence(0, todoPage.getTotalPages() - 1)}">
      <span th:if="${i == todoPage.getNumber()}" th:text="${i + 1}"></span>
      <a th:if="${i != todoPage.getNumber()}"
         th:href="@{/todo/query(page=${i})}" th:text="${i + 1}"></a><!-- ② -->
    </li>
    <li>
      <span th:if="${todoPage.isLast()}">次→</span>
      <a th:unless="${todoPage.isLast()}"
         th:href="@{/todo/query(page = (${todoPage.getNumber()+ 1}))}">次→</a><!-- ② -->
     </li>
  </ul>
</div>
```

まずページリンクの上位div要素にページリンクを表示するかどうかの判断を追加します(①)。

ここでページリンクが不要なのは、以下の場合です。

・検索条件にエラーがあった場合 → コントローラーはtodoListオブジェクトにnullを設定する。
・該当するToDoが無かった場合 → todoListの要素数が0(nullではない)
つまり「todoList == null　または　todoListの要素数 == 0」ならページリンクは不要です。よってこの逆(否定)である「todoList != null　かつ　todoListの要素数 != 0」の場合、ページリンクを表示させます。

> ある条件式の否定形を求めるとき「ド・モルガンの法則」を知っていると便利です。
> ■ド・モルガンの法則
> NOT(X AND Y) = NOT(X) OR NOT(Y)
> NOT(X OR Y) = NOT(X) AND NOT(Y)
> X,Yは条件式(booleanを返す)、NOTは否定を表すとします。
>
> ここではXをtodoList == null, YをtodoListの要素数 == 0とします。すると最初のページリンク不要の条件式はX AND Yとなります。これの否定NOT(X AND Y)は、上記法則より NOT(X) OR NOT(Y)となります。これでX,Yを元に戻すと
> NOT(todoList == null) OR NOT(todoListの要素数 == 0)
> です。NOTを外して簡単にすると以下のようになります。
> todoList != null OR todoListの要素数 != 0
>
> このテクニックは
> ・if文のthen句とelse句の内容を入れ替えてプログラムを読みやすくしたいとき
> ・early returnにするとき
> などに役立つことがあるので、覚えておいて損はないです。

次は【ルート3】の各ページリンクをクリックしたときに対応するものです。
このリンクには、次に表示するページをコントローラーに知らせる情報が必要です。そのためページリンクの「←前」、ページ番号、「次→」のリンク先を/todo/query?page=nに変更します(②)。
これを以下のハンドラーメソッドで処理します。

【リスト11-9】
com.example.todolist.controller.TodoListController.java(step4;一部抜粋)

```
  @GetMapping("/todo/query")
```

```
  public ModelAndView queryTodo(@PageableDefault(page = 0, size = 5)
Pageable pageable,
                                ModelAndView mv) {
    mv.setViewName("todoList");

    // sessionに保存されている条件で検索
    TodoQuery todoQuery = (TodoQuery)session.getAttribute("todoQuery");
    Page<Todo> todoPage = todoDaoImpl.findByCriteria(todoQuery, pageable);

    mv.addObject("todoQuery", todoQuery); // 検索条件表示用
    mv.addObject("todoPage", todoPage); // page情報
    mv.addObject("todoList", todoPage.getContent()); // 検索結果

    return mv;
  }
```

[検索]ボタンクリック時のハンドラーメソッドとほぼ同じですが、検索条件はフォームではなくセッションに保存されているものを使います(GETリクエストではフォーム内容を取得できない)。またセッションに検索条件があるということは、入力チェックでエラーが無かったことになるので、入力チェックも省いています。

演習課題

ここまで作ってきたToDo管理には、残念ながら少々使い勝手の悪いところがあります。実務でであれば、対応すべき項目のように思われます。いずれも本書で説明してきた内容で解決できますので、チャレンジしてみてください。

(実装例はサポートサイトより入手できます)

①更新の場合、完了日が過去でもエラーにしない

完了日が過ぎたToDoを更新しようとすると「期限を設定するときは今日以降にしてください」というエラーメッセージが表示されます。そのため期限オーバーのToDoは件名、重要度、緊急度だけ直すことができません。そこで更新の場合、完了日はyyyy-mm-ddの形式チェックだけするよう変更してください。

②入力画面から元のページへ戻る

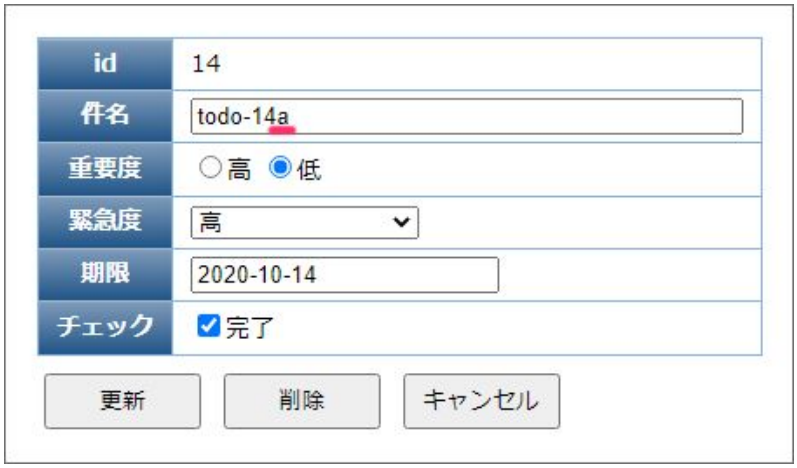
入力画面で[登録][更新][削除][キャンセル]ボタンを押下すると、1ページ目に戻ります。これを入力画面へ遷移する前に表示していたページへ戻るよう変更してください。

例．ToDo更新の場合

現状)

①3ページ目のtodo-14をクリック

②件名を「todo-14a」に変更して[登録]ボタンをクリックする。



③表示は1ページに戻る。

新規追加

id	件名	重要度	緊急度	期限	完了
1	todo-1	★	★	2020-10-01	
2	todo-2	★	★★★	2020-10-02	完了
3	todo-3	★★★	★	2020-10-03	
4	todo-4	★★★	★★★	2020-10-04	完了
5	todo-5	★	★	2020-10-05	

1 / 5 ページを表示中

←前 1 2 3 4 5 次→

変更後)

①3ページ目のtodo-14をクリック

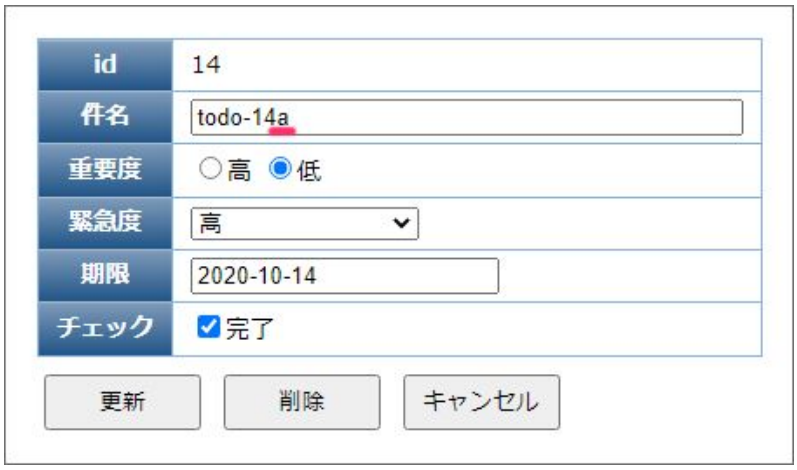
新規追加

id	件名	重要度	緊急度	期限	完了
11	todo-11	★★★	★	2020-10-11	
12	todo-12	★★★	★★★	2020-10-12	完了
13	todo-13	★	★	2020-10-13	
14	todo-14	★	★★★	2020-10-14	完了
15	todo-15	★★★	★	2020-10-15	

3 / 5 ページを表示中

←前 1 2 3 4 5 次→

②件名を「todo-14a」に変更して[登録]ボタンをクリックする。



③入力画面へ遷移する前に表示していた3ページ目に戻る。変更箇所が確認できる。

③ページリンクの範囲を制限する

ページリンクを表示中ページの前後2ページ分だけにしてください。

現状)たとえば、検索結果が12ページあったら、1～12がページリンクになる。

変更後)ページリンクを以下のようにする。

表示中ページ	ページリンク範囲	ページリンク全体(下線はリンクを表す)
1	1～3	←前　1　2　3　次→
2	1～4	←前　1　2　3　4　次→
3	1～5	←前　1　2　3　4　5　次→
4	2～6	←前　2　3　4　5　6　次→
:	:	:
9	7～11	←前　7　8　9　10　11　次→
10	8～12	←前　8　9　10　11　12　次→
11	19～12	←前　9　10　11　12　次→
12	10～12	←前　10　11　12　次→

参考資料

書籍

この本を読んだ後、さらにSpring Bootを学びたい方には、以下の3冊をお薦めします。

ただしいずれもSpring Boot3ではなく、SPring Boot2用なので留意してください。

(書籍情報は2022年12月時点のもの)。

Spring Boot 2 入門: 基礎から実演まで(Kindle版)

作者:原田 けいと,竹田 甘地,Robert Segawa / 発売日:2020/05/22 / 価格：980円

★次に読むならこの本。本書では扱っていない項目、あるいは同じ項目でもまた別の角度から解説されており、理解度を深めることができます。

Spring Boot 2 プログラミング入門

作者:掌田津耶乃 / 発売日: 2018/1/30 / 価格：3,080円(単行本), 2,772円(Kindle版)

★レベルアップを目指すならこの本。「オリジナルのバリデーターを作る」など、Spring Bootの使いこなす上で有用な情報が多数書かれています。ただ掲載されているプログラムリストが見にくいのと、文章が少々わかりにくいのが残念。

Spring徹底入門 Spring FrameworkによるJavaアプリケーション開発(大型本)

作者:株式会社NTTデータ / 発売日: 2016/07/21 / 価格：4,400円(大型本), 3,960円(Kindle版)

★Spring Bootのベースとなっている Spring Frameworkに関する書籍。Spring Bootの根本原理を理解したいならこの本は欠かせません。本格的にやるなら手元に置いておきたい1冊。ただし「徹底入門」とあるが入門者用ではない。ある程度知っている人のための本です。

サイト

インターネット上にも数多くの情報源があります。Spring Boot関連で日頃筆者がよく利用させてもらっているのは、以下のサイトです。

Spring Boot
https://spring.io/projects/spring-boot
★Spring Boot開発元のサイト。

Qiita
https://qiita.com/
★プログラマー向け技術情報共有サービス。このなかにSpring Bootに関する記事も数多く含まれている。ただし内容は高度なものが多い印象。

StackOverflow
https://stackoverflow.com/　英語
https://ja.stackoverflow.com/　日本語
★プログラミングに関するQ&Aサイト。英語版は圧倒的なボリュームを持つ。エラーメッセージをキーにしてGoogleで検索すると、ここにたどり着くことが多い印象。

TERASOLUNA Server Framework for Java (5.x) Development Guideline

https://terasolunaorg.github.io/guideline/5.5.1.RELEASE/ja/index.html#

★TERASOLUNAは、株式会社NTTデータの開発している比較的規模が大きなシステム開発手順、フレームワーク、サポートのブランド名です。「TERASOLUNA Server Framework for Java」はオープンソース化されたフレームワークでSpring Frameworkを使っています。このガイドラインはSpringの知識だけでなく、Webアプリケーションを構築する上で示唆に富む内容を数多く含んでおり参考になります。

英語のサイトも多いですが、ChromeでGoogle翻訳の拡張機能でページ全体を翻訳すると大体の意味はつかめます。意味不明なところはDeepL(https://www.deepl.com/ja/translator)を使うと、良い結果が得られることもあります。

奥付

菊田 英明(きくた ひであき)

Java言語と出会ったのは1995年の終わりごろ。JDKはまだβ版だった。当初は「趣味」でJavaプログラムを書いていたが、いつのまにか仕事もJava一色となる。その後はWebアプリケーションシステムの開発に従事する。某エンジニアリング会社勤務を経て2019年4月より個人事業主。近年は新入社員向けJava導入教育の講師も請け負っている。

■保有する資格

情報処理技術者試験

プロジェクトマネージャ

アプリケーションエンジニア

プロダクションエンジニア

データベーススペシャリスト

オンライン情報処理技術

基本情報処理技術者

Sun Certified Programmer for the Java Platform

■著書

「実践 JDBC—Javaデータベースプログラミング術 」(オーム社)

「SE・プログラマスタートアップテキストJSP 基礎」(技術評論社)

「基本情報技術者 らくらく突破 Java」(共著、技術評論社)
「Spring Bootで始めるWebアプリケーション開発入門」(Kindle)
「Spring Bootで始めるWebアプリケーション開発入門(実践編)」(Kindle)
「Spring Bootで始めるWebアプリケーション開発入門(実戦編)」(Kindle)

表紙デザイン：後藤あゆみ

Spring Boot3で始めるWebアプリケーション開発入門(基礎編)

2022年12月05日　初版発行

著者　　菊田英明
発行者　菊田英明